GPU Implementation of Scanline Stereo With Application to Stereoscopic Video Enhancement

Jason Eveleth[2] Ethan Williams[12]
Brown University
CSCI 1290[1] APMA 2822B[2]

Abstract

Creating high-quality stereographic video from a low-quality pair of video streams is a potent task in computational photography, with the potential to democratize the recording of virtual reality experiences. We present a performant application, which takes in a stereoscopic video stream, computes a dense matching, learns a color transformation and outputs a unified stream. We discuss our CUDA and OpenMP implementations of scanline stereo. We thoroughly profiled and optimized these implementations and present our application.

1. Problem Setting

This project seeks to derive a coherent stereoscopic video from a stereo pair of cameras that independently select exposure and color balance settings.

With the increasing access to virtual reality headsets, the capability to view stereoscopic video is growing rapidly. Correspondingly, there is an increasing demand for the production of stereo video.

Ideally, stereo video could be captured with inexpensive USB webcams. These devices have problems that hinder their suitability for capturing stereoscopic video, however. The most inexpensive webcams available do not have the capability to adjust exposure or color balance. Worse, they produce heavily compressed, independently noisy output.

Computational tools that correct for these shortcomings could dramatically widen access to the capability to create stereoscopic video experiences.

1.1. Extension to Other Problems

The tools created during the course of this project have straightforward applications to a number of other photography problems. Instead of outputting a coherent stereo pair, it would be trivial to output a wide field-of-view video by including the non-overlapping portion of each video. By only selecting the intersecting portion of the feeds, a high quality denoised output is obtained. These single feed output options are useful for high quality video conferencing, where rather than buying a high quality video camera, the fusion of two noisy, inexpensive cameras would produce an acceptable result.

If we consider random noise in the cameras video stream as an independent random variable, we could combine the two streams to lower the total variance. Additionally, if the cameras are compressing their output, and the compression artifacts occur independently, we can use differences to eliminate those as well.

2. Solution Strategy

There are four phases to our solution.

1. Calibration: We rectify the images so that the rows of the image have the same y-value in the scene.

2. Matching: We determine a dense matching for each column of the left (source) image to the corresponding column in the right (destination) image.

3. Correction: We learn a function from source image color to destination image color.

4. Unify: We output the stereo pair, with the destination image corrected.

3. Calibration: Rectify the Input Pair

Sparse point correspondences are calculated using scikit-image's implementation of SIFT. Points are filtered using RANSAC, and the fundamental matrix was calculated, also with scikit-image. OpenCV's `stereoRectifyUncalibrated` is used to derive homography transformations to rectify the source image pair.

This approach is not successful. Even in ideal conditions with static pre-rectified images, the rectification is unstable. A variety of implementations of SIFT and the fundamental matrix solver were tested. The authors presently believe the issue to be due to a known bug

in `stereoRectifyUncalibrated`, as example code from OpenCV, and elsewhere, suffered from the same instability. A long term solution would probably involve using fiducial markers to gather higher quality point correspondences. An undistortion step is also warranted.

4. Corruption: Mock the Data

An approach was needed to address the lack of rectified stereo video from actual webcams. As a temporary measure, stereoscopic video was gathered from YouTube [5], and artificially corrupted to reproduce the problem setting.

The first step in the data corruption process is the color transformation. Users are given the ability to tweak the color using sliders for lightness, tint, and color temperature. These labels are given to the process of scaling and offsetting the input pixel colors after they have been transformed into the L*a*b* color space.

The next step is the addition of Gaussian noise. Each input pixel is given a offset based off of a continuously changing Gaussian noise map. The standard deviation of the noise profile is user configurable. This is intended to simulate sensor noise.

As a final step, the input images are compressed using JPEG, and immediately uncompressed. After altering the input colors and applying the Gaussian noise map, we hope to develop unique compression artifacts, as would be seen in the output of low quality webcams. In an alternative implementation, the compression parameters could be made user-configurable.

5. Matching: Scanline Stereo

The scanline stereo algorithm is uniquely suited toward finding dense pixel matches suitable for this problem.

Since our input is two rectified images, the rows are views of the same height in the scene. Thus, when trying to match the images, we can focus on a single row when we are trying to compute the matching. We want to find the "best" map from source column to destination column along that row. We defined optimality by computing a divergence matrix (see Figure 1 and Figure 2) for all the columns in the source to all the columns in the destination, and finding the minimum cost path along that matrix.

More concretely, let's say we're calculating the correspondence of row r. We will define the divergence between columns c_s and c_d as the mean squared error between the patches surrounding the pixel at r, c_s in the source image and the pixel at r, c_d in the destination image (this is clearly visualized in Figure 1). We will refer to the MSE between two pixels to be the pixel divergence, and the MSE between two patches (the sum of pixel divergences) to be the patch divergence. This gives us a measure of patch divergence centered at the two pixels. If we arrange those divergences into a matrix (like in Figure 2), a path through that matrix is a correspondence between source and destination columns.

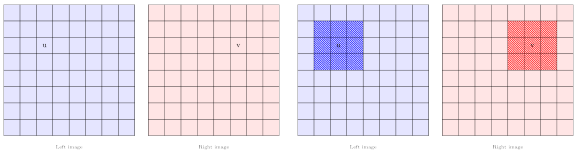


Figure 1. A visualization of pixel and patch divergence centered at u and v. Note: they are in the same row, but in different columns. In this paper we work with each row independently.

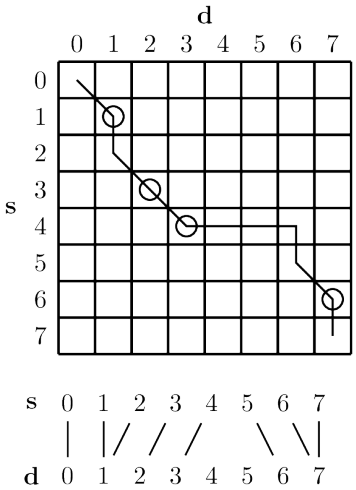


Figure 2. Fixing a row, the goal of the algorithm is to find a correspondence between every column in a scanline in the source image with a column in the corresponding scanline in the destination image. We do this by tracing a minimum cost sequence, starting from the last column in the source and destination indices.

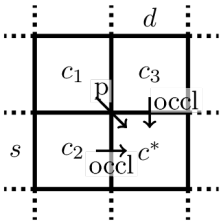


Figure 3. We want to find the cost of c^*. This calculation is independent for each row, so let r be the row. We define p to be the patch divergence at r, s, d, and *occl* to be the occlusion penalty. Then $c^* = \min(c_1 + \text{p}, c_2 + \text{occl}, c_3 + \text{occl})$.

We calculate the cost of a path through this matrix of patch MSEs recursively as either an occlusion penalty along the source or occlusion penalty along destination columns or the patch divergence between the two pixels. See Figure 3 for more details.

We want to find the min cost path, which will correspond to the optimal correspondence between source and desti-

nation columns. We can do that using a technique called "dynamic programming". This involves storing intermediate results in a table to calculate the minimum path. This allows us to compute the min cost path.

So, to complete Stage 2, (1) we compute the divergence matrix for each row, (2) we solve for the min path for each row, (3) we output that paths' column correspondences.

5.1. Scanline Stereo Stages

5.1.1 Pixel Divergence

The pixel divergence, pixel$[r, s, d]$, is a measure of the difference between the pixel in the source image at r, s, and the pixel in the destination image at r, d (as seen in the left subfigure of Figure 1). This project uses the squared difference of the luminance values. Other implementations may choose an alternative to squaring that's less sensitive to outliers. They may also choose the negative product of the pixel intensities, potentially inspired by convolution. This is straightforwardly implemented as triply-nested for loops.

5.1.2 Patch Divergence

The patch divergence, patch$[r, s, d]$, is a measure of the difference between the patch of images surrounding r, s in the source image, and r, d in the destination image (as seen in the right subfigure of Figure 1). This quantity is a sum of the pixel similarities of the pixels in the region $r - P, s - P$, to $r + P, s + P$. The parameter P is configured by the user as needed. This may be implemented by triply nested for loops, over r, s, d. The neighboring pixels that make up the patch may then be iterated over, and a final sum of the patch divergence is totalled and stored.

5.1.3 Costs

The costs tensor, cost$[r, s, d]$ represents the quality of the match between the pixels r, s and r, d. This divergence measure includes the disparities of the pixels with lesser column indexes, in the same scanline, in the best match. This is recursively implemented, as shown in Figure 3. The terms of the minimum represent the correspondence case, the source occlusion case, and the destination occlusion case. Note that the cost is only a function of the cost of the pixels in the same scanline, which gives the scanline stereo algorithm its name. This stage may be implemented using dynamic programming [1]. The cost computation for pixels with $r + s = k$ only depend on the pixels with $r' + s' = k - 1$, meaning that each diagonal can be successively computed once the previous diagonal has been computed.

5.1.4 Traceback

The traceback stage is responsible for computing a correspondence map, and a valid pixels map. The correspondence map, correspondence$[r, s]$, represents the optimal column that was found in the cost computation. The valid map, valid$[r, s]$ represents whether the path in costs was achieved by a correspondence; whether the first argument was selected as the minimum. This can be computed by tracing a path back from the cost matrix, with max indices, to the origin position. Each step in the minimum should be selected by visiting the pixel above, to the left, and diagonally up, at each index.

5.1.5 Fast Patch Similarity

Consider computing the patch divergence between a pair of patches in one image; and computing the divergence between a similar pair, one row down. These patches are identical, excepting the the bottom row in the first pair, and the top row in the second pair. Also consider a pair of patches, each shifted over by a column. These patches are also very similar to the patches in the first pair. Our implementation is able to intelligently avoid this repeated work [1] [6].

We take inspiration from the differential prefix sum technique for computing the sum of a subsequence of an array. We re-write the naive nested summation as a constant number of lookups in a prefix sum tensor, prefix$[r, s, d]$. This allows us to effectively save work across rows and columns. Let pixel$[r, s, d]$, abbreviated p, be a measure of the divergence of the pixels at r, s in the source image, and r, d in the destination image.

$$
\begin{aligned}
&\text{prefix}[r, s, d] \\
&= \sum_{r'=0}^{r} \sum_{s'=0}^{s} \text{pixel}[r', s', s' + d - s] \\
&\text{patch}[r, s, d] \\
&= \sum_{r'=r}^{r+P} \sum_{s'=s}^{s+P} \text{pixel}[r', s', s' + d - s] \\
&= \sum_{r'=0}^{r+P} \left(\sum_{s'=0}^{s+P} p - \sum_{s'=0}^{s} p \right) - \sum_{r'=0}^{r} \left(\sum_{s'=0}^{s+P} p - \sum_{s'=0}^{s} p \right) \\
&= \sum_{r'=0}^{r+P} \sum_{s'=0}^{s+P} p - \sum_{r'=0}^{r+P} \sum_{s'=0}^{s} p - \sum_{r'=0}^{r} \sum_{s'=0}^{s+P} p + \sum_{r'=0}^{r} \sum_{s'=0}^{s} p \\
&= \text{prefix}[r + P, r + P, d + P] - \text{prefix}[r + P, s, d] \\
&- \text{prefix}[r + P, s + P, d + P] + \text{prefix}[r, s, d]
\end{aligned}
$$

A straightforward implementation of $\sum_{r'=r}^{r+P} \sum_{r'=r}^{s+P} \text{pixel}[r', s', s' + d - s]$ requires a number of operations proportional to the square of the patch size,

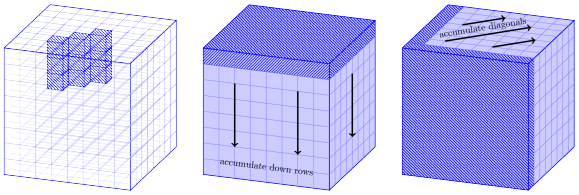


Figure 4. The first image shows what a pair of image patches looks like in the pixel$[r, s, d]$ tensor. The second two images describe the prefix sum procedure: the prefix sum is first computed along the rows, then diagonally along the columns.

while the new expression involves a constant number of lookups, sums, and differences.

This however, relies on the prefix being computed in its entirety. These entries can be computed using a cumulative sum technique. Each entry can be re-written as a sum of previously computed entries: prefix$[r, s, d]$ = patch$[r, s, d]$ + prefix$[r-1, s, d]$ + prefix$[r, s-1, d-1]$. Using this recursive dependency, we can avoid computing the sum explicitly for every r, s, d, saving work. Instead, we implement this using two prefix sums: one over the rows, and one diagonally, over the columns. A complete analysis of the number of floating point operations and memory usage required for this approach is included in a later section.

Each stage of the scanline stereo algorithm is presented below. Pseudocode is included for the purposes of this analysis. Integer operations have been elided, edge cases are not considered, and only the innermost statements of each stage are shown. Let R, S, D, and P be the number of rows in the images, number of columns in the source image, number of columns in the destination image, and the number of pixels in the patch, respectively.

The pseudocode and arithmetic intensity computations are independent of the implementation, which is why this analysis is included before the details are discussed.

5.1.6 Pixel Divergence

```
double distance = src[r, s] - dst[r, d];
pixel_divergence[r, s, d] = distance * distance;
```

This sample is run once for every row, column in the source image, and column in the destination image. We perform two loads, one subtraction, one product, and one store. However, the `src` and `dst` arrays are of size $RS + RD$, times 8 bytes. With optimal caching, they could all be read into memory exactly once. With the loads and stores amortized, the arithmetic intensity becomes AI$= \frac{2RSD}{RS+RD+RSD} \frac{\text{flops}}{8 \text{ bytes}}$

The total number of floating point operations is $2RSD$. The total memory traffic is $RS + RD + RSD$, times 8 bytes.

5.1.7 Prefix Sum, Along the Rows

```
for (long r = 1; r < rows; r++) {
    array[r, s, d] += array[r - 1, s, d];
}
```

This sample is run once for every column in the source image, and column in the destination image. We load a value from the array, perform an add with the accumulated total, and store the result back to the array. The arithmetic intensity is AI $= \frac{1}{2} \frac{\text{flops}}{8 \text{ bytes}} = \frac{1}{16} \frac{\text{flops}}{\text{byte}}$. The total number of floating point operations is RSD. The total memory traffic is $2RSD \times 8$ bytes.

5.1.8 Prefix Sum, Diagonally Along the Columns

```
while (s < cols_src && d < cols_dst) {
    array[r, s, d] += array[r, s - 1, d - 1];
    s++; d++;
}
```

This sample is run once for every row, and column in the source image. We load a value from the array, perform an add with the accumulated total, and store the result back to the array. The arithmetic intensity is AI $= \frac{1}{2} \frac{\text{flops}}{8 \text{ bytes}} = \frac{1}{16} \frac{\text{flops}}{\text{byte}}$. The total number of floating point operations is RSD. The total memory traffic is $2RSD \times 8$ bytes.

5.1.9 Naive Patch Divergence

```
for (long rn = r - p; rn <= r + p; rn++) {
    for (long sn = s - p; sn <= s + p; sn++) {
        long dn = sn + d - s;
        sum += pixel_divergence[rn, sn, dn];
    }
}
patch_similarity[r, s, d] = sum / count;
```

This sample is run once for every row, column in the source image, and column in the destination image. We assume optimal caching, so each float from the `pixel_divergence` array is only loaded once, even though we use it in multiple iterations. Thus, in total we load and store RSD 8-byte values. We do $RSD(2P+1)^2$ additions and one division. Note that the count is a constant equal to $(2P+1)^2$ in non edge cases. The arithmetic intensity is AI $= \frac{RSD((2P+1)^2+1)}{2RSD} \frac{\text{flops}}{8 \text{ bytes}} = \frac{(2P+1)^2+1}{16} \frac{\text{flops}}{\text{byte}}$. The total number of floating point operations is $RSD((2P+1)^2+1)$. The total memory traffic is $2RSD \times 8$ bytes.

5.1.10 Fast Patch Divergence

```
double sum = pixel_divergence[rp, sp, dp]
    + pixel_divergence[rm, sm, dm]
    - pixel_divergence[rp, sm, dm]
    - pixel_divergence[rm, sp, dp];
patch_divergence[r, s, d] = sum / count;
```

This sample is run once for every row, column in the source image, and column in the destination image. Using the same logic from the previous step, we load 1 value (amortized), perform 3 adds, a division, and store the result back. Note that the count is a constant equal to $(2P+1)^2$ in non edge cases. The arithmetic intensity is AI $= \frac{4}{2}\frac{\text{flops}}{8\text{ bytes}} = \frac{1}{4}\frac{\text{flops}}{\text{byte}}$. The total number of floating point operations is $4RSD$. The total memory traffic is $2RSD \times 8$ bytes.

5.1.11 Cost Computation

```
double correspond = cost[r, s - 1, d - 1] +
    patch_divergence[r, s, d];
double occlusion_src = cost[r, s - 1, d] +
    occlusion_cost;
double occlusion_dst = cost[r, s, d - 1] +
    occlusion_cost;
cost[r, s, d] = fmin(correspond, fmin(
    occlusion_src, occlusion_dst));
traceback[r,s,d] = fargmin(correspond,
    occlusion_src, occlusion_dst);
```

We have two helper functions: fmin which calculates the min of two floats, and fargmin which return 0 1 or 2 depending on which float is the smallest. This sample is run once for every row, column in the source image, and column in the destination image. We load 2 8-byte values (amortized load from cost, and one load from patch_divergence), perform 3 adds, two minimums (we optimally reuse the results of the fmin operation in the fargmin), and store the result back, and the direction we used (1-byte value). The arithmetic intensity is AI $= \frac{5RSD}{3RSD\times 8 + RSD\times 1}\frac{\text{flops}}{\text{byte}} = \frac{1}{5}\frac{\text{flops}}{\text{byte}}$. The total number of floating point operations is $5RSD$. The total memory traffic is $25RSD$ bytes.

5.1.12 Traceback

```
while (s != 0 && d != 0) {
    long argmin = traceback[r, s, d];
    s -= {1, 1, 0}[argmin];
    d -= {1, 0, 1}[argmin];
    if (argmin == 0) {
        correspondence[r, s] = d;
        valid[r, s] = 1;
    }
}
```

This sample is run once for every row. We load one value, two stores, and no perform no floating point operations. The arithmetic intensity is therefore zero. The number of iterations is is bounded by the sum of the number of columns in the source and the destination images. The total memory traffic is $9RS + R(S+D)$ bytes.

5.2. Performance Analysis

A CPU implementation of the scanline stereo algorithm, with fast patch similarity, is provided as part of this project. Support for multi-threading was added using OpenMP.

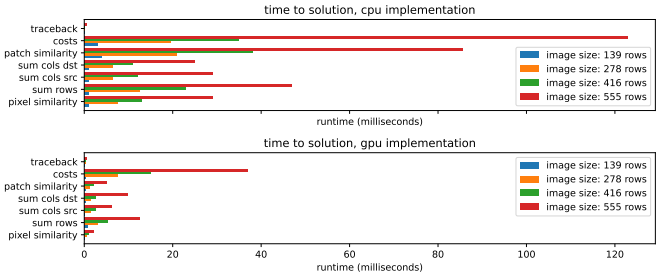


Figure 5. Time to solution for the CPU and GPU implementations, naive patch similarity kernel not included. The Bowling1 image was selected, with a `patch_size=12,occlusion_cost=10**-3`. A variety of image sizes are included, but the rows are labelled. Each sample has identical aspect ratio.

A CUDA implementation of the scanline stereo algorithm is also provided. Computation is parallelized across the rows, source columns, and destination columns, as applicable. The cost computation features parallel computation across column diagonals. Significant effort was made to reduce branching operations and to re-use memory.

5.2.1 Time to Solution

In order to deliver a real-time product, the time to solution must be below a threshold. This is the amount of time required to create a dense disparity map, given an input pair of stereo images. Only the fast GPU and CPU implementations are considered for this analysis, as the naive implementations are intractable for real-time performance. The runtime of each stage was plotted for a variety of image sizes 5. Assuming a fixed aspect ratio, there is a roughly cubic relationship between the image height, and the runtime for each of the stages (except traceback). The GPU implementation is much quicker across all of the stages. To collect this data, the median of 20 trials was recorded at each image size.

5.2.2 Nsight Systems Profile

In order to analyze the performance of the GPU implementation, screenshots have been provided of the Nsight Systems profiler, 6, 7. These screenshots show several trials, scheduled back-to-back. The runtime of each stage and the relative runtime of the kernels is shown visibly. The profile also shows the memory operations.

From the profiler screenshots, it can be seen that copying data between the host and device represents an insignificant amount of the time-to-solution. The amount of data being processed by the kernels is on the order RSD bytes, while the data being transferred is on the order of $RS + RD$ bytes.

From the screenshots, you can also see the relationship between the patch size and the runtime of the algorithm. As the patch size grows, the runtime of the fast kernel does

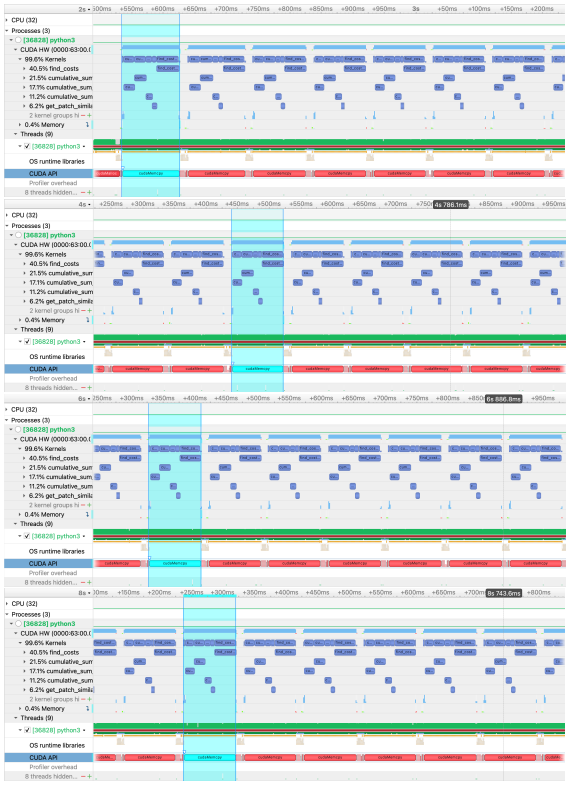


Figure 6. Fast solution Nsight profile. Screenshots for patch sizes 1, 3, 9, and 30 are shown.

Figure 7. Naive solution Nsight profile. Screenshots for patch sizes 1, 3, 9, and 30 are shown.

not change. With a large patch size, the naive kernel's patch similarity finder dominates the runtime of the entire solution. The runtime also grows quadratically with the size of the patches.

5.2.3 Roofline and FLOP Rate

Experiments were conducted using on a Quadro RTX 6000 (GPU implementation), and a Intel Core i7-1165G7 (CPU implementation), so roofline analysis was conducted for these systems 8. We choose to conduct a roofline analysis for a `patch_size=3`, on a 370 pixels by 417 pixel stereo pair.

Experiments were conducted with a stereo pair of 370 by 417 images: the third size Bowling1 [7] image. Timings were gathered by taking the median across 20 trials, and the timings were plotted against the arithmetic intensity of each stage. The traceback phase could not be included in the figures, since it has zero arithmetic intensity.

The naive patch similarity presents a significant outiler. In this phase, $(2P+1)^2$ iterations of a nested for loop are conducted, and there is significant overlap in the memory usage between iterations. With an optimal iteration order and a large cache, theoretically each value has to be read once from the pixel similarity tensor. In this way, it has a very large arithmetic intensity.

There is a significant limitation in our analysis. Our count of the number of floating point operations does not consider edge cases. For large images, there are few patches on the edge of the image, so the effect should be limited. However, with large patch sizes, it can be the case that we over-count floating point operations. This led to a bug where the naive patch similarity was above the roofline in an earlier iteration of the chart. A more accurate estimate of the floating point operation count would lead to points being slightly below their current positions.

None of our stages achieve the maximum flop rate allowed by the roofline.

This may be partially due to optimistic specifications for our memory bandwidth and floating point operation rate. Other assumptions about are hardware are also infeasible, including the optimal caching assumption, which is impossible on real hardware.

The overall runtime of each stage is dependent on more than just floating point operations. Some stages, especially the fast patch similarity stage, rely significantly on integer operations. This decreases the overall time to solution.

The cost calculation, and prefix sum stages have significant data dependencies. Before some entries can be computed, we rely on the results from previous entries. For ex-

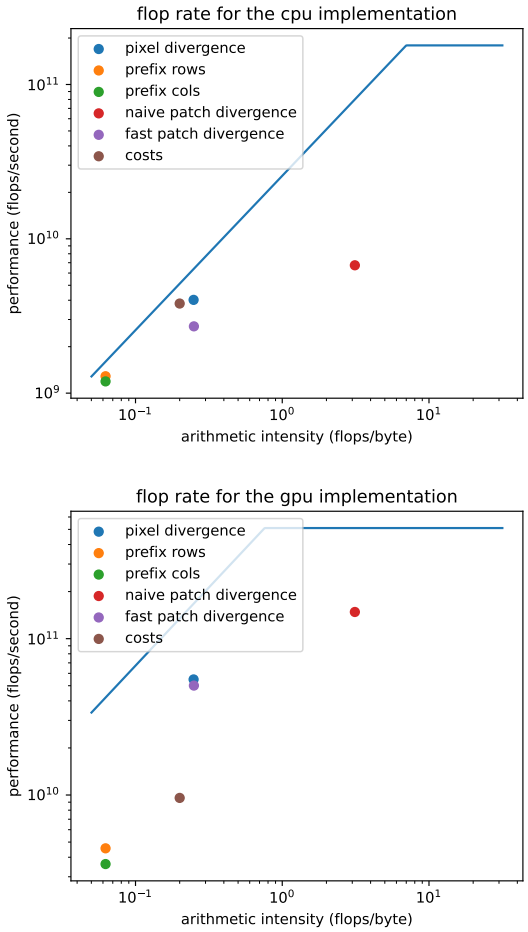


Figure 8. Roofline chart for the CPU and GPU implementations.

ample, before a diagonal in costs is computed, we have to wait until all of the threads complete the previous row. This effects the performance in two ways: we may not be able to use the full hardware parallelism, and the synchronization instructions have additional overhead.

Branching has an impact on our performance. In the GPU implementation, since our images may not have dimensions that are multiples of the thread block size, conditionals are required to avoid out-of-bounds writes. There are conditionals deeply embedded in the logic of the cost stage especially. On the GPU, conditionals can cause warp divergence, where operations in a warp are serialized. This can prevent the full utilization of hardware capabilities. The CPU implementation has many fewer of these kind of conditionals.

We are memory bound, with the exception of naive patch similarity. The naive patch similarity becomes arbitrarily compute bound as the patch size grows. More calculations are needed on the same amount of data.

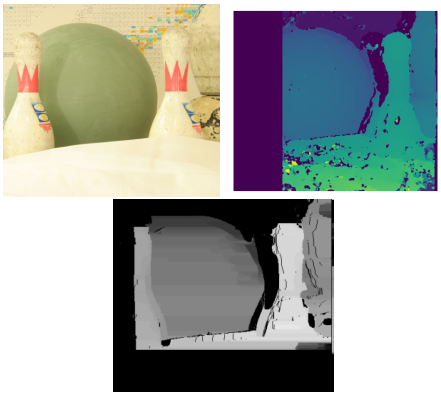


Figure 9. Comparison of StereoSGBM with the results from scanline stereo. The first image is the left image in the stereo pair. The next image is result from OpenCV. The last image is our result.

5.3. Comparison with Related Work

For the purposes of comparison, we ran our test image, 370 by 417 pixels through OpenCV's StereoSGBM matcher [3]. This algorithm is fundamentally different than the one we have chosen to implement; however it is a frequently used real-time disparity map estimator. We achieved comparable results with `numDisparities=100, blockSize=11`. The time-to-solution was 26.04 milliseconds. Our solution, running on the GPU, with `patch_size=30, occlusion_cost=0.001`, completes in 84.01 milliseconds. The CPU solution, parallelized with OpenMP, completes in 788.1 milliseconds. Results with more disparity resolution and fewer incongruities take longer with StereoSGBM. Tweaking parameters of our algorithm has a negligible impact on performance, as they only impact the traceback stage.

The quality of the results are significantly different between the two implementations. For example, our algorithm is completely unable to find the disparity for the pixels on the table in the scene. The table has little texture, so our algorithm is unable to discriminate between these patches. StereoSGBM does not suffer from this problem as it produces results that are coherent across rows. Our algorithm produces results that are acceptable for the bowling ball and bowling pin, as patches on these surfaces are textured enough to provide meaningful inter-patch discrimination.

5.4. Discussion of Optimizations

5.4.1 Removing the Patch Divergence Intermediate

It was investigated whether removing the patch divergence intermediate from the algorithm would improve performance. Instead of having a separate patch divergence stage, the patch divergence code could instead be inlined into the cost stage kernel. This was attempted, and there was a significant performance regression. Both strategies have identical flop counts, so it may be initially surprising that removing a large inter-

mediate would actually reduce performance. We consider the most likely explanation that the memory access pattern of on pixel$[r, s, d]$ in the cost stage is less predictable than accessing it during a separate patch similarity stage. As we traverse the diagonals during the cost stage, the relevant entries in the pixel$[r, s, d]$ tensor appear almost random, being in arbitrary rows and columns.

5.4.2 Removal of the Cost Intermediate

In order to produce the output valid and correspondence maps, we do not need an explicit cost intermediate. Instead, we can store integers representing the path back to the output. We call this tensor traceback. Instead of storing the full double precision values, only a few bits are needed to represent the next step in the path. These saves memory.

However, in order to compute the an entry of the traceback tensor, the costs above, to the left, and diagonally up from the entry need to be stored as seen in Figure 3. Our implementation stores three diagonals of the cost tensor in shared memory, which satisfies the memory dependency, while avoiding unnecessary allocations and slow reads and writes to main memory. This change improved our time to solution significantly.

5.4.3 Single Precision Floating Point Numbers

Calculations on single precision floating point numbers can be much quicker than calculations on double precision floating point numbers. In addition, single precision floating point numbers require less memory to store, reducing the memory traffic.

It is uncertain whether single precision floating point numbers are viable to use with the fast patch similarity algorithm. They have very limited precision, at 23 bits, or roughly 7 digits. Our algorithm relies on summing across the rows and columns of the source images, meaning that the maximum value we need to store is on the order of several hundred thousand. Precision is significantly limited when storing numbers of this magnitude. In order to compute the patch similarity, we intend to subtract values of the array. When we subtract two numbers of this size, floating point cancellation causes our estimations of the patch similarity to become highly inaccurate. These issues compound as input image dimensions become larger. With the much more precise double precision floating point numbers, these issues are avoided.

5.4.4 Future

With more time, there are several strategies that we predict would improve performance.

The implementation included in this project for prefix sum is naive. We selected the simplest implementation in order to ensure accuracy. As the project developed, we recognized that the prefix sum stages represented a much smaller fraction of the runtime than the cost stage, so optimization effort was not directed towards prefix sum. We recognize there are a variety of parallel prefix sum techniques. It is ambiguous whether they would improve performance: our parallelism is already saturated by the fact that we need to compute several independent prefix sums. It is worth investigation, however.

Another strategy worth investigation is to round the image size up to be a multiple of the thread block size. By guaranteeing that each thread is processing a valid pixel, it would be possible to remove several conditionals. This approach uses memory however. Care must be taken to avoid making these boundaries a part of the result.

Another strategy worth investigation is to change the way we calculate the denominator of the patch similarity (the pixel `counts`). For most of the patches, the `count` is $(2P + 1)^2$, so the `count` computation could be broken up into different kernels, or computed incrementally, and this would save some time. However, computing `count` did not represent a large fraction of the overall time to solution, so this was not investigated.

6. Correction: Color Mapping

The goal of the color mapping stage is to make the destination image look like the source image. Idealistically, the mapped destination image and the original source image could be delivered as the output of the entire pipeline. This stage suffers from inherent issues, so it serves as a first-pass output. It has the benefit of being complete however: every pixel in the destination image is transformed.

The input to this stage are the two input images, and the dense, high quality pixel correspondences from the scanline stereo stage. The respective pixel values are extracted from the correspondences. The pixel values are converted into the $L^*a^*b^*$ color space. A linear mapping from each source color channel to the corresponding destination channel is derived. For this project, a linear mapping is assumed to be acceptable.

The inverse of the model is applied to the destination image. The result is a recolored destination image that is visually similar to the source image.

6.1. Limitations

There are several issues with this approach. Highlights and underexposed regions represent parts of the image where the true color could not be derived. Very bright and very dark pixels are manually filtered when creating the model, but the model is naively applied to these highlights to generate the output. The output is unrealistic in some regions for this reason. A more careful approach might preserve highlights and underexposed regions when generating the output.

Noise, including compression artifacts and sensor noise, represent outliers for this approach. The intention is that with enough pixel correspondences, the noise will cancel out. A more robust approach would take more care when developing the model, and use an approach like RANSAC.

The true mapping from the source colors to the destination colors is, in general, not linear. An alternative implementation might generate a per-channel lookup table.

Despite the limitations, the problem setting assumes that the source and destination feed have largely similar properties. This naive color mapping approach delivers a first-pass.

7. Unify: Make Enhanced Stereoscopic Video Stream

The goal of the unification stage is to deliver a coherent stereo video. The destination image is altered to look like the source image. The inputs to this stage are the source image, the input destination image, color mapped destination image, and the dense pixel correspondences.

A spatially mapped image is generated by borrowing the pixels from the source image according to the dense correspondences. This image has features in the same location as the corresponding features in the destination image, however it is made from source image pixels. This means that the color profile matches identically with the source image.

This image is missing pixels in the places where the scanline stereo algorithm was unable to determine correspondences. These holes are filled in with pixels from the first-pass, color mapping stage.

The result is a transformed version of the destination image that has the same color properties as the source image. This destination image is output alongside the source image.

7.1. Limitations

A more sophisticated implementation would use an alternative scheme to compose the color mapped and spatially mapped images. This would remove the seams that are sometimes visible in the output. It would also help smooth the color transitions.

8. Application

In order to demonstrate the pipeline, an application was written. This application takes a stereoscopic video as input, and outputs an enhanced stereoscopic video.

The performance of the application, as run on the authors' laptops, is very poor. With a `rows=320` video, the application runs at approximately 1 fps. The scanline stereo algorithm represents the majority of the runtime for each frame. The authors do not have access to a device with a GPU, so the CPU implementation of the scanline stereo algorithm must be run at this degraded speed.

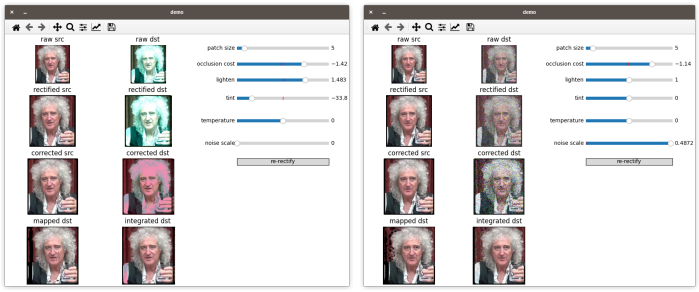


Figure 10. First: dst features an extreme brightening, tint, and no injected noise. Second: dst image features extreme noise injection.

A screenshot of the application 10 is included to highlight the capabilities and limitations.

The top row represents the input images. The reader should distinguish the left (src) and right (dst) images by the fact that Brian's head is positioned more to the left in the src image. The right image, labelled dst, is artificially brightened and tinted. This is to simulate each camera independently adjusting exposure and color balance. These corruption parameters were chosen using the lighten and tint sliders.

The next pair is the rectified images. Since the images are already rectified, and our rectification code is flawed, no transformation is performed.

Next is the color corrected image pair. Since the highlights are extreme in the dst image, the color correction failed, and the red from the curtain was applied to the hair and shirt. These unrealistically extreme settings were chosen to demonstrate the integration image (discussed below).

The next row contains the mapped and the integrated images. The mapped image is the spatially mapped image that borrows pixels from src to reproduce dst. As you can see, Brian's head is positioned to the right, as in dst. However, the color is natural, as in src. There is a notable gap in the left of the image where no corresponding pixels in src could be found. The mapped image can be altered by adjusting the parmeters of the scanline stereo algorithm with the patch size and occlusion cost sliders. The integrated image is formed by composing the mapped image and the corrected dst image. The left of Brian's shirt has the artifacts from corrected dst.

A second screenshot is provided with an extreme level of injected noise 10. An output dst image recovered with acceptable quality.

Larger screenshots are provided in a supplemental page. The resulting stereoscopic video streams are very high quality, despite the extreme corruption.

Corrupting both the src and the dst images has not been attempted, but likely models the problem setting more accurately.

9. Division of Effort

9.1. Jason

For the code, I helped design and write the CUDA kernels. I helped design and write a prototype color and exposure correction program.

For the report, I drafted the intro, and wrote many of the diagrams. I checked the performance analysis sections, and made some small general edits to the paper.

9.2. Ethan

Initially drafted the correction model with Jason.

I also helped design and write the GPU implementation. I ported the GPU code to the CPU using OpenMP. I created the gallery, and the performance graphics.

I wrote the bindings to call the C++ and CUDA code from Python.

Did the initial flop count and arithmetic intensity calculations. Wrote much of the discussion sections.

After we finished the version of the report submitted APMA 2822B, I did the remaining work. This includes writing the application, and the write up of the pipeline in the report, and modifying the presentation slides for computational photography.

10. Gallery

We processed each of the images in the Middlebury 2006 dataset [7] with our scanline stereo implementation. Each triplet includes the left image in the stereo pair `view1.png`, the ground truth disparity map `disp1.png`, and our result. These results were achieved with fixed parameters across all of the images `patch_size=7,occlusion_cost=10**-2.1`.

[2] [4]

References

[1] Ingemar J. Cox et al. "A Maximum Likelihood Stereo Algorithm". In: *Computer Vision and Image Understanding* 63.3 (May 1996), pp. 542–567. ISSN: 1077-3142. DOI: 10.1006/cviu.1996.0040. URL: http://dx.doi.org/10.1006/cviu.1996.0040.

[2] M. Domnguez-Morales et al. "Stereo Matching: From the Basis to Neuromorphic Engineering". In: *Current Advancements in Stereo Vision*. InTech, July 2012. DOI: 10.5772/45901. URL: http://dx.doi.org/10.5772/45901.

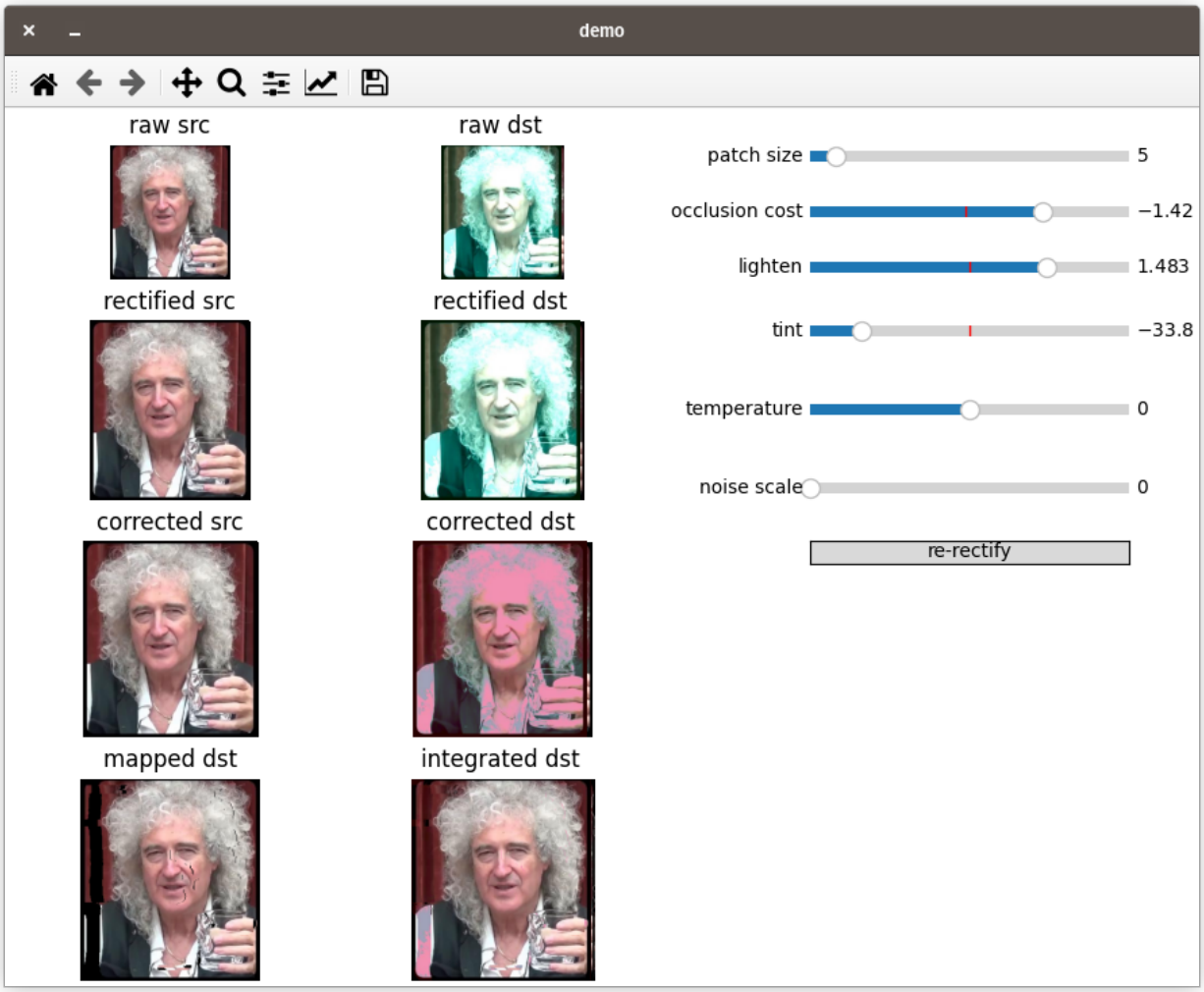
[3] Heiko Hirschmuller. "Stereo Processing by Semiglobal Matching and Mutual Information". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.2 (2008), pp. 328–341. DOI: 10.1109/TPAMI.2007.1166.

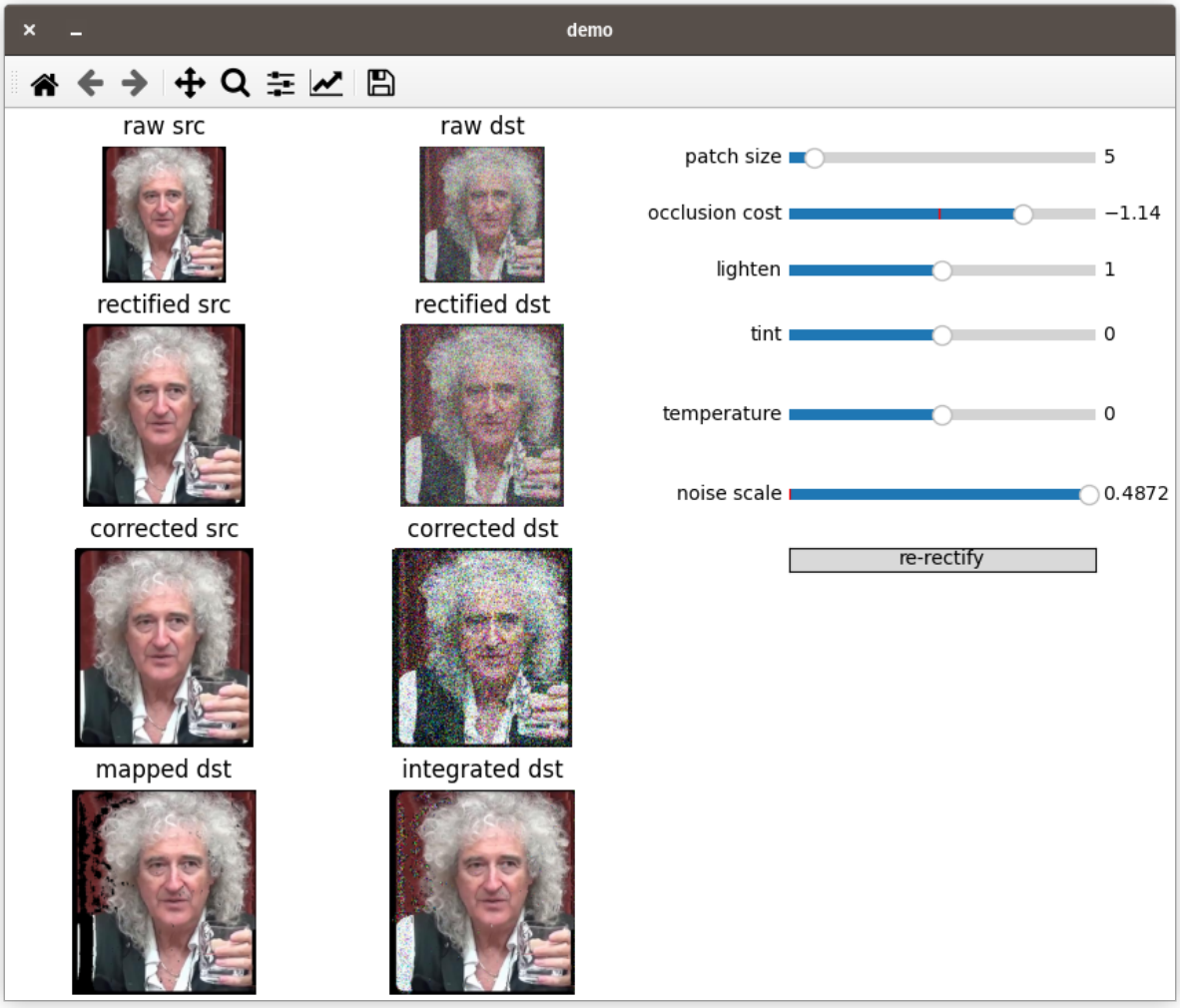
[4] Neeraj Krishna. *A Comprehensive Tutorial on Stereo Geometry and Stereo Rectification with Python — towardsdatascience.com*. https://towardsdatascience.com/7f368b09924a. [Accessed 13-12-2023].

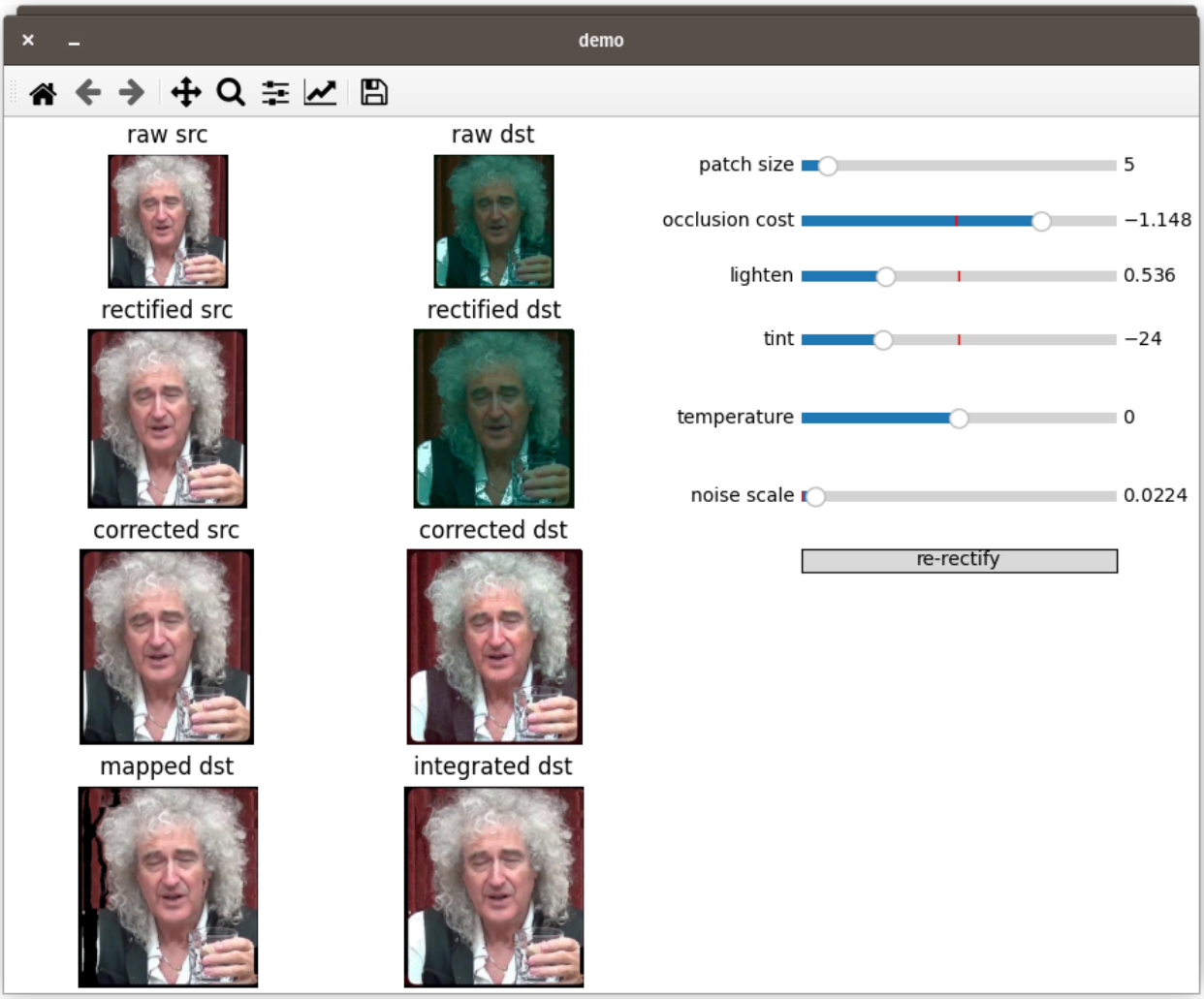
[5] Brian May. *Brian May - Brian Takes Stereo Photos with a Smartphone*. Youtube. 2016. URL: https://www.youtube.com/watch?v=0mWoiJPytco.

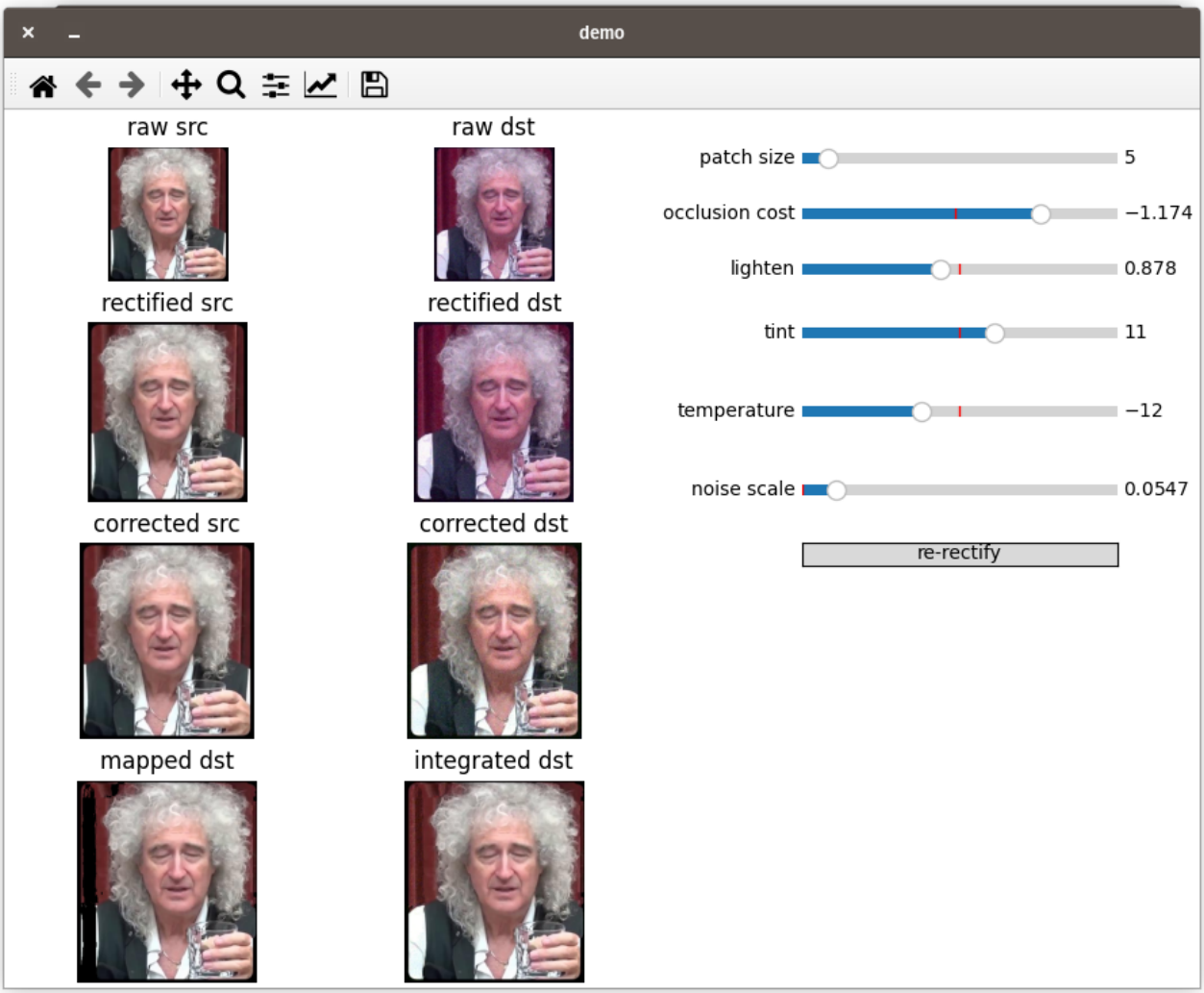
[6] Yuichi Ohta and Takeo Kanade. "Stereo by Intra- and Inter-Scanline Search Using Dynamic Programming". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-7.2 (1985), pp. 139–154. DOI: 10.1109/TPAMI.1985.4767639.

[7] Daniel Scharstein and Chris Pal. "Learning Conditional Random Fields for Stereo". In: *2007 IEEE Conference on Computer Vision and Pattern Recognition*. 2007, pp. 1–8. DOI: 10.1109/CVPR.2007.383191.


demo
raw src
raw dst
rectified src
rectified dst
corrected src
corrected dst
mapped dst
integrated dst
patch size
5
occlusion cost
−1.42
lighten
1.483
tint
−33.8
temperature
0
noise scale
0
re-rectify


demo
raw src
raw dst
rectified src
rectified dst
corrected src
corrected dst
mapped dst
integrated dst
patch size
5
occlusion cost
−1.14
lighten
1
tint
0
temperature
0
noise scale
0.4872
re-rectify


demo
raw src
raw dst
rectified src
rectified dst
corrected src
corrected dst
mapped dst
integrated dst
patch size
5
occlusion cost
−1.148
lighten
0.536
tint
−24
temperature
0
noise scale
0.0224
re-rectify


demo
raw src
raw dst
rectified src
rectified dst
corrected src
corrected dst
mapped dst
integrated dst
patch size
5
occlusion cost
−1.174
lighten
0.878
tint
11
temperature
−12
noise scale
0.0547
re-rectify