STEREO MATCHING WITH PATCHMATCH

Niklaus Bamert, Jordan Burklund, Thomas Etterlin

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Due to their inherently parallel nature, many image processing algorithms lend themselves well to high-performance implementations based on SIMD operations. We demonstrate a 8.9x speedup over a C++ baseline implementation of the PatchMatch [1] stereo matching algorithm on the Intel Haswell architecture.

1. INTRODUCTION

In the field of computer vision, the two-view stereo matching problem consists of determining feature correspondences between two images L and R that have been acquired with a stereo camera. With these correspondences it is possible to make inferences about the depth of each matched feature point in the scene. In *dense* stereo correspondence matching, the goal is to generate a depth estimate for each pixel in the scene. Such stereo algorithms are a common in robotic vision pipelines for object avoidance, autonomous driving or 3D scene reconstruction for example.

Feature correspondences can occur between arbitrary points $p \in L$ and $p' \in R$. A brute-force correspondence search scales quadratically in the number of pixels, making a naive solution approach to this problem computationally very expensive. Image features are often non-unique due to homogeneous surfaces, specularities, or lighting differences between the two camera views, and features may only be visible in one of the two views as a result of occlusion by other objects.

These issues are commonly dealt with by extracting features around a given point p in an image that are robust against the common photo-metric challenges, such that re-identifying them is more likely.

Related Work. For dense stereo matching, such extracted features are often based on an image patch of considerable size [2, 1, 3], making these approaches computationally expensive. The work by Zhang et al. [3] is closely related to [1], however it has some issues reconstructing slanted surfaces as a result of an initial matching step relying on discrete, rather than continuous, disparities. In [2], Gallup et al. use slanted surfaces throughout their implementation in order to reach sub-integer disparity estimates. Similarly, in [1], slanted surface estimates are used as a means to estimate continuous disparities. Furthermore, the approach lends itself to optimization, since the operations within the cost window are inherently parallel.

In the following sections we introduce the dense stereo matching algorithm *Patch Match* [1], published by Rhemann et al., analyze it for its optimization potential, and report results on the achieved 8.9x speedup.

2. BACKGROUND

In this section we first formally define the rectified stereo matching problem. In further subsections, the *Patch Match* stereo algorithm will be described, followed by a cost analysis.

In the rectified stereo matching problem we assume we are given two images of the same scene as input. These images are assumed to be captured from slightly different view points at the same instant of time. For simplicity, but w.l.o.g, we further assume that both cameras are mounted facing the same spatial direction, with their view-points only differing by a horizontal baseline-shift b. Under a geometric transformation called *image rectification*, we can ensure that corresponding features in both images are located on the same horizontal pixel line. We call the distance of the x-coordinate at which we find corresponding features of the left image L in the right image R the *disparity*. Given the disparity d, the baseline-shift b and the focal length of the cameras f, the distance z of a given feature from the camera can be determined as follows

$$z = \frac{fb}{d}. \tag{1}$$

The correspondence search space under image rectification reduces the algebraic complexity of the stereo matching problem from $\mathcal{O}((HW)^2)$ to $\mathcal{O}(HW^2)$. Here we have used W and H for the image width and height, respectively.

The author thanks Jelena Kovacevic. This paper is a modified version of the template she used in her class.

Patch Match Stereo. The Patch Match stereo algorithm offers smooth disparity estimates based on slanted support windows. These slanted support windows are enabled by storing a plane proposal $f_p = (a_f, b_f, c_f)$ for any given pixel p of the images L and R. The plane proposal f_p for a pixel p has a direct relationship to the disparity estimate as follows

$$d = a_f x + b_f y + c_f \tag{2}$$

where (x, y) are the 2D image coordinates of pixel p.

Patch Match starts by randomly initializing the planes f_p of each pixel. The disparity estimation is then iteratively refined by propagating the plane estimates for each pixel in a *Spatial* propagation step based on the immediate pixel neighbors, and a *View* propagation step that propagates plane estimates from one image to the other. These two propagation steps are followed by a *Plane* refinement step which adds small, random perturbations to the plane estimate of a pixel in an attempt to improve the estimate. Note that the original Patch Match paper [1] presents an additional *Temporal* propagation step. This step is only relevant when working with stereo image sequences, such as videos, and is not considered for this project.

Both the *Spatial* and *View* propagation steps of Patch Match exhibit a spatial dependency on the current plane estimates. Additionally, there is a temporal dependency among all propagation steps over several iterations of the algorithm preventing any parallelization. Common to all three propagation steps is the evaluation of a window-based cost function:

$$\text{cost}(p, f_p) = \sum_{q \in W_p} w(p,q) \cdot \rho(q, \underbrace{q - (a_f q_x + b_f q_y + c_f)}_{=:q'}). \tag{3}$$

Here p is any given pixel in either L or R and W_p is a square window centered at p of size 35×35 pixels. Additionally $q' = q - (a_f q_x + b_f q_y + c_f)$, which is the current correspondence estimate for q based on the plane estimate f_p.

$$w(p,q) = e^{-\frac{\|I_p - I_q\|_1}{\gamma}} \tag{4}$$

The w term shown in Eq. (4) is a weighting function that returns high values if pixels p and q in the same image I are similar. Pixels are compared by the l_1-norm in RGB space and γ is a tuning parameter which has been fixed to $\gamma = 10$ by the authors of the Patch Match paper [1].

$$\begin{aligned}\rho(q, q') = (1-\alpha) \cdot \min(\|I_q - I_{q'}\|_1, \tau_{col}) \\ + \alpha \cdot \min(\|\nabla I_q - \nabla I_{q'}\|_1, \tau_{grad})\end{aligned} \tag{5}$$

The ρ term in Eq. (5) captures the pixel dissimilarity between q and q' based on intensity difference both in the RGB values I and the image gradient ∇I. In [1], the authors fix α, τ_{col} and τ_{grad} to 0.9, 10 and 2, respectively. The data required by both w and ρ is visualized in Fig. 1.

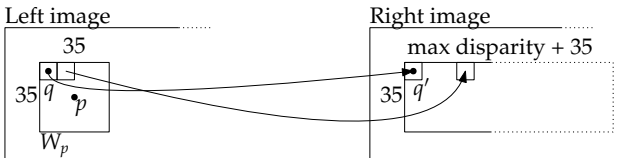

Fig. 1: Cost function operation visualized. w from Eq. (4) is based on a fixed cost window and can be precomputed. ρ from Eq. (5) is data dependent and can have erratic memory access patterns due to discontinuous disparity estimates q'.

Each of the three refinement steps operates on each pixel in the image over several iterations. At each iteration, the refinement steps are applied to each pixel starting from the upper left and progressing row-wise to the bottom right, and repeating again from the bottom right and progressing row-wise to the top left of the image.

The algorithm is initialized by precomputing the image gradients for both left and right images, and initializing the plane parameters such that the corresponding plane normals are of unit length in a random direction. After the refinement steps are applied, the post-processing step applies several filters to smooth out invalid pixels and account for regions of occlusion. Pixels with invalid plane parameters are filled in with a weighted mean filter. A short summary of the algorithm relevant to our work can be found in Algorithm 1.

Algorithm 1 PatchMatch Stereo algorithm

```
1: procedure PATCHMATCHSTEREO
2:     PREPROCESSING
3:     for 6 iterations do
4:         for all pixels p with plane estimate f_p do
5:             SPATIALPROPAGATION        ▷ calls cost(p, f_p)
6:             VIEWPROPAGATION           ▷ calls cost(p, f_p)
7:             PLANEREFINEMENT           ▷ calls cost(p, f_p)
8:     POSTPROCESSING
```

Cost Analysis. Although both the initialization and post-processing steps are important to the results of the algorithm, the associated runtime for both steps is relatively insignificant compared to the refinement steps and were not considered for optimization. Profiling the rest of the algorithm revealed that 99% of the execution time of the algorithm is spent within the cost function of Eq. (3). Furthermore 52% of the execution time is spent on computing the exponential function in Eq. (4) which is called inside of the cost function. This makes the cost function an appealing primary target for optimizations.

A cost analysis was performed for the baseline implementation to gain a better understanding of where to focus optimization efforts. Only the refinement steps were considered, since they take the majority of the runtime. The costs we considered were all floating point operations, and

	W(operations)	Q(bytes)
cost	$(48 + W_{exp})w^2$	$29w^2$
Spatial	$2W_{cost}$	$160 + 2Q_{cost}$
View	$6cols + W_{cost}$	$48cols + Q_{cost}$
Plane	$W = 225 + 9W_{cost}$	$207 + 9Q_{cost}$

Table 1: Cost analysis of baseline functions. Here $cols$ is the width of the full image, w is the window size, and W_{exp} is the operational cost of `std::exp()`. Note that these are costs for each pixel, and all costs are implicitly multiplied by the image width and height.

included the integer operations for computing the image intensity differences. Although these operations are not typically combined, the cost analysis will give a good approximation of where the bulk of the computation and memory bandwidth usage occur. The number of operations W and memory traffic Q in bytes is shown in Table 1.

From the cost analysis, it is clear that the cost function dominates the operational intensity of each of the refinement steps, and gives further justification to focus entirely on optimization of the cost function. This also shows that the operational intensity of the cost function will always be the main limiting factor for each of the refinement steps for most window sizes and any image size. From Table 1 and Fig. 3 we can see that the problem is bounded by computation rather than memory bandwidth.

3. IMPLEMENTATION

In the following we present a subset of the implemented optimizations that led to the most significant speed-ups or gave us valuable insights. For the source code we refer to [4].

Baseline. The baseline implementation is based on [5] and all dependencies except for `libpng` were removed. By removing OpenCV and using simple arrays, any overhead of containers was eliminated and enabled changes to the underlying data structures. The structure of the code follows Algorithm 1.

For the validation of our implemented methods, we use a unit-test and standard evaluation metrics for dense two-frame stereo correspondence algorithms, such as the average disparity error. In contrast to the entire algorithm, the cost function is deterministic and the main target for optimization. Thus we implemented a unit-test that compares the output of our various optimizations to the baseline output. Due to possible algebraic transformations, a relative tolerance of 0.002 was introduced which can lead to different execution paths in the algorithm. To account for this potential change of the final output, the Middlebury evaluation framework [6] was used for validation. All optimizations pass the unit-test and lead to similar or better results in the Middlebury evaluation compared to the baseline.

Exponential Function. For the baseline implementation, the C++ function `std::exp()` was used to calculate the exponential for the weights in Eq. (4) of the cost function. From the runtime analysis of the baseline implementation, this function takes 52% of the runtime, and is a strong candidate for optimization. This function also has platform and compiler specific implementations that make it difficult to compute the operational intensity.

Functions that approximate the exponential value were considered, but ultimately a table-lookup based implementation was favored due to the reduced amount of computation required. The exponential function is only required for the cost function, and can be optimized for this specific use case. The operand of the exponential is the l_1-norm of the image channel intensity differences which is bounded in the range $[0, 765]$, and is in the set of integers. Thus, a finite size lookup-table can be created and indexed by the l_1-norm.

The table values can be computed in two ways: either at compile-time, or at run-time. A code generator was used to generate a static array that could be precomputed by the compiler, and was compared to generating the values for the lookup table during initialization at run time. Both methods achieved similar execution times during the table lookup, and generation of the table at run-time took an insignificant amount of time to initialize. The runtime based generation was selected for the final implementation since it was easier to parameterize, easier to maintain and took an insignificant amount of time to initialize.

Since the weight of every pixel in a window remains constant these weights can also be precomputed. Rather than computing the weights of each pixels during initialization and using a significant amount of memory, the weight of each pixel q in the window is precomputed before the refinement steps are applied to pixel p and reused by multiple calls to the cost function.

Single Instruction Multiple Data (SIMD). As stated in Section 2, we are not bound by memory bandwidth and can use a SIMD implementation to gain additional performance. As an initial step for the SIMD implementation, slight adjustments to the code were performed. All conditional statements were replaced by binary masking operations. The interleaved 8-bit RGB storage format posed a potential problem since 24 bits were stored per pixel. To reduce the necessary shuffle operations we added an additional unused alpha channel to the pixel such that we had a 8-bit RGBA storage format using 32 bits per pixel. The usage of a power of two storage size greatly simplified the implementation and had no performance drop but padding the data reduced the effective SIMD usage to $3/4$. As a target platform we picked the Haswell microarchitecture with AVX2 and FMA instructions. Note that we only mention the main insights gathered

in the process of implementing SIMD.

The baseline implementation uses a double loop to calculate the cost over the 35x35 window. Since a majority of the calculations use floating point numbers, we can process 8 pixels of the same row simultaneously using the full 256 bits of the SIMD lanes. Since windows have the a width of 35 pixels, we can only get a maximal effective SIMD usage of $^{35}/_{40}$. As depicted in Fig. 1, the corresponding pixel locations in the right image can be scattered over the row of the image, and this access can be quite erratic due to the random nature of the plane estimates f_p. This means that slower gather functions must be used to load the image data.

The first significant SIMD version is labeled `simd_v3`. We refrain from showing implementation details on this non-final version and refer the reader to the source code. Note that due to the storage format (8-bit RGBA), we need to convert the image intensities to floats (32-bit) and multiple shuffles are needed in this process.

Since an updated cost analysis showed that we are still not bound by memory bandwidth, we decided to reduce the amount of shuffle and conversion instructions by storing the image in a 32-bit float RGBA format that uses 128 bits per pixel. We call this implementation `simd_v10`.

Reordering of instructions. Between `simd_v10` and `simd_v17`, some of the instructions were reordered to improve the instruction pool usage and hide latencies. With `simd_v10`, the difference of gradients was computed after the difference of RGB intensities but in `simd_v17`, these computations are in the reverse order. This allows hiding some of the latencies of loading the data and computation of the image gradient cost behind the beginning of the RGB computations. The compiler and CPU instruction prefetcher should be intelligent enough to do this automatically, but explicitly changing the order in the source code resulted in a slight boost in performance.

Split RGB channels. Since the interleaved RGBA format leaves $^1/_4$ of the SIMD lane unused and requires multiple shuffles, we considered other storage formats. We tested a de-interleaved format where each color channel is stored in a separate array and is labeled `simd_v21`. This implementation removes the need for the alpha channel, and uses the SIMD lanes more effectively.

4. EXPERIMENTAL RESULTS

In this section we present the maximum speedup we achieved, present a roofline analysis and show that our implementation is within 2x of the maximal performance considering the critical path of the cost function. We furthermore show that our code scales well for relevant image sizes.

For all experiments throughout our work we used an Intel Xeon E5-2680v3 processors with 2.5 GHz, 32 KB L1-cache, 256 KB L2-cache and 30 MB SmartCache (L3). Throughout the presented results we used the GNU compiler (`gcc version 6.3.0 (GCC)`) with the following flags: `g++ -std=c++11 -O3 -march=native -mavx2 -mfma`. No performance gain over GCC was observed during experiments with `icc` and different optimization flags.

All of the experiments were performed on image data from the Middlebury data set. Due to the run-time dependency the algorithm on a particular image, we always used the Teddy image of the 2003 dataset [7] for timing-critical experiments. Similar results were observed with the other images of the dataset. If not stated otherwise, the quarter sized image is used with dimensions of 375×450 pixels.

Speedup. Figure 2 shows the speedup for all of the implemented kernels. In total we have 4 major improvements.

The first is the optimization of the exponential function and precomputation of the weights (`precomp_fast_exp`), giving a speedup of 2.65x compared to the baseline.

The second is the first SIMD iteration (`simd_v3`) giving a speedup of 2.4x and total speedup of 6.3x with respect to to the baseline. Note that the introduction of the additional alpha channel (see `rgba`) did not reduce performance, since the algorithm is not bounded by memory bandwidth.

The third improvement (`simd_v10`) gives a speedup of 1.25x and a total speedup of 7.9x. Due to the reduction of conversions and shuffle operations, the increase in memory usage as a result of the use of 32-bit floats instead of 8-bit integers did not adversely affect performance.

The last minor optimization (`simd_v17`) gives us the final speedup of 8.9x.

In contrast to our expectations, `simd_v21` reduced the overall speedup to 5.8x despite using SIMD lanes more effectively. One possible explanation is the increased number of gather operations required. Instead of using in total 8 `_mm256_loadu2_m128` in the loop body we have 6 `_mm256_mask_i32gather_ps` operations that are needed which load 32-bit at 48 possible locations rather than 128-bit at 16 different locations. In other words, the single R, G, B arrays are packed less efficiently for the access method that the algorithm requires.

Roofline Model. In the following we provide an analysis with the roofline model and establish some runtime bounds on our problem.

For the creation of the roofline model, the RDTSC instruction cycles were used to show performance, while the data and operations were obtained from a manual analysis of the individual kernels. From our analysis we expect to be limited by the L3 cache for the given image size (375×450). Our instruction mix however, does not perfectly fit the FMA instructions, so an upper bound on performance is more complex than just the SIMD FMA peak performance.

As previously mentioned in Section 2 we can see in Fig. 3 that the baseline is beyond the ridge point and the algorithm isn't limited by memory bandwidth. The implementation

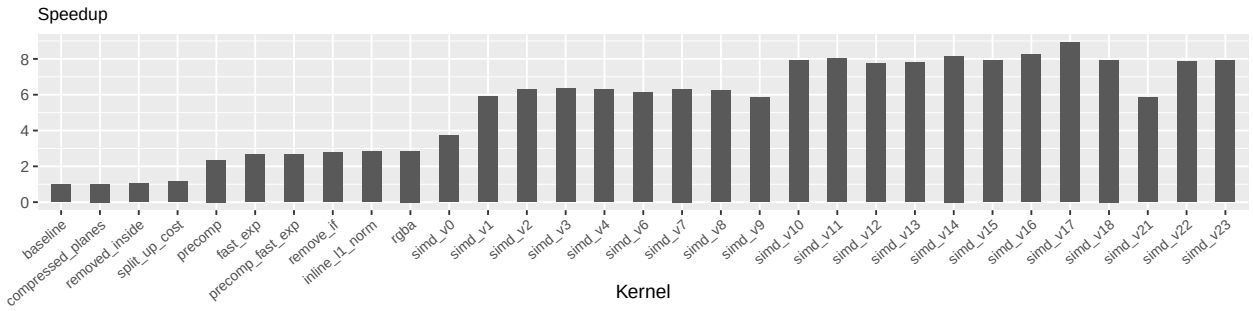

Fig. 2: Speedup of all implemented kernels relative to the baseline, runtime of baseline 493s, 375x450 Teddy image

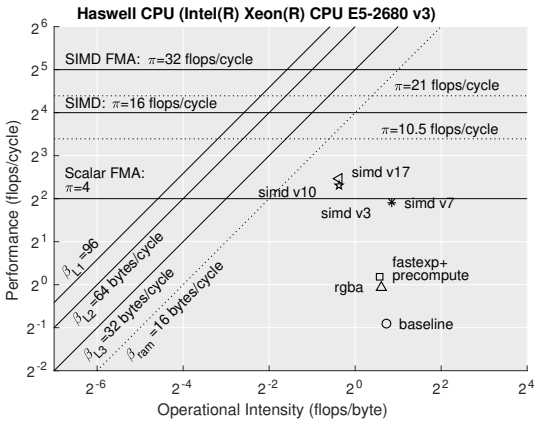

Fig. 3: Roofline plot with mentioned kernels. The dotted performance bounds represent the reduced performance due to unused SIMD channels and the limiting window size.

with the precomputation of the weights and fast exponential function reduces the operational intensity and increases the performance to 1.1 flops/cycle.

The first SIMD iteration (`simd_v3`) gets 4.8 flops/cycle. By using 32-bit floats as image storage instead of 8-bit integers, the operational intensity further decreases for `simd_v10` and `simd_v17`. Due to a reduction in other operations we get a maximal performance of 5.5 flops/cycle for `simd_v17`.

In order to quantify how close we are to the maximal possible performance, we will present multiple bounds to show that our implementation is pushing the limits. The first and obvious bound is the peak performance of 32 flops/cycle for SIMD FMA of which we reach 17.3%. Taking into consideration that our algorithm forces us to use a window size of 35x35 we will always have some unused SIMD lanes and the maximal usage is thus 35⁄40 for every row of the window. Furthermore the introduction of an unused alpha channel decreases the peak performance by another ¼. Thus we get a reduced peak performance of 21 flops/cycle, of which we reach 26.4%. Note though that this reduced peak performance is not a hard bound since other data layouts and/or computation orders could potentially perform better. Nevertheless our tests showed that a removal of the alpha channel and de-interleaved data storage did not lead to a further increase in performance.

For a tighter upper bound we analyzed the latency of the critical path in our computations in the cost function. Only the necessary computations in the evaluation of the cost function were considered. Any type conversions, integer computations and others were ignored. In addition to these perfect conditions we assumed that enough ports are always available and that all the data resides in L1-cache which is highly unlikely. Figure 4 shows the dependency graph of the computation of one pixel in the window of the cost function. We want to remark that unrolling of the loop could invalidate this bound. Nevertheless, unrolling of the loop did not lead to any performance increase which we are attributing to the fact that the loop body contains many instructions. We see that the critical path has a latency of 36 cycles and with a window size of 35x35 and 8 computations in a SIMD lane we get a lower bound on the cost function of 5500 cycles. Thus we get 47% of the maximal performance with respect to this bound.

Variable Problem Sizes. Figure 5 shows the influence of different sized working sets on the performance of three different kernels. The average cycle count for the cost function was recorded for rescaled versions of the Teddy image starting from a size of 150×180 up to the full resolution of 1500×1800. Depending on the kernel used, a different amount of data is used for the same image size. Note that already for the smallest size of 150x180 the data does not fully fit into L2 cache anymore. We see that for small sizes we get a substantially better performance than for bigger sizes. This effect can be partly attributed to the fact that boundary effects come into play. For small image sizes the cost window is often smaller than 35x35 and less computations have to be performed. Even when the problem size is bigger than the L3 cache, we do not get a substantial performance drop and the cycle count for our problem is nearly independent of the problem size.

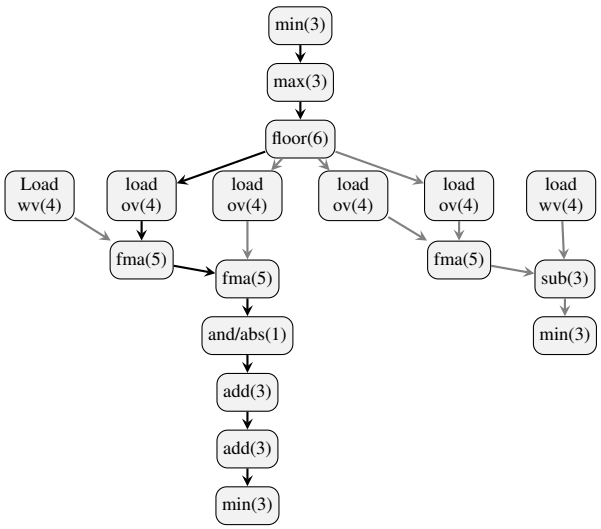

Fig. 4: DAG of the dependencies for the cost calculation of one pixel, numbers in braces denote the latencies of the operations. Black arrows depict the critical path. wv stands for the working view and ov for the other view where we find the correspondences.

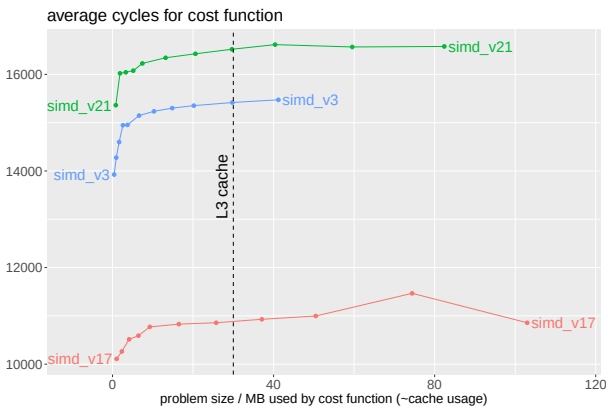

Fig. 5: Cycle count of various implementation depending on the memory by the cost function which range comes from image sizes ranging from 150x180 up to 1500x180.

5. CONCLUSION

We proposed a method to eliminate the expensive exponential weighting term for discrete image intensity differences. We also showed how the erratic access pattern in the corresponding view can still be mapped to a SIMD implementation. Note that for the special case of fronto-parallel windows, the expensive gather operations could be replaced by aligned loads and result in a substantial speedup. We achieved an overall speedup of 8.9x from the baseline, and concluded our investigation by showing that we are within a factor of approximately 2 of the optimal performance considering instruction latencies for the Haswell architecture.

6. REFERENCES

[1] Christoph Rhemann Michael Bleyer and Carsten Rother, "Patchmatch stereo - stereo matching with slanted support windows," in *Proceedings of the British Machine Vision Conference*. 2011, pp. 14.1–14.11, BMVA Press, http://dx.doi.org/10.5244/C.25.14.

[2] D. Gallup, J. Frahm, P. Mordohai, Q. Yang, and M. Pollefeys, "Real-time plane-sweeping stereo with multiple sweeping directions," in *2007 IEEE Conference on Computer Vision and Pattern Recognition*, June 2007, pp. 1–8.

[3] Y. Zhang, M. Gong, and Y. Yang, "Local stereo matching with 3d adaptive cost aggregation for slanted surface modeling and sub-pixel accuracy," in *2008 19th International Conference on Pattern Recognition*, Dec 2008, pp. 1–4.

[4] Thomas Etterlin Niklaus Bamert, Jordan Burklund, "Patch match stereo implementation," `https://gitlab.ethz.ch/bamertn/pmstereo`, 2019.

[5] Ivan Bergonzani, "Patch match stereo implementation," `https://github.com/ivanbergonzani/patch-match-stereo`, 2018.

[6] Daniel Scharstein and Richard Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International journal of computer vision*, vol. 47, no. 1-3, pp. 7–42, 2002.

[7] Daniel Scharstein and Richard Szeliski, "High-accuracy stereo depth maps using structured light," in *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.* IEEE, 2003, vol. 1, pp. I–I.