sensors

MDPI

Article

ReS^{2}tAC—UAV-Borne Real-Time SGM Stereo Optimized for Embedded ARM and CUDA Devices

Boitumelo Ruf [1,2,*], Jonas Mohrs [1], Martin Weinmann [2], Stefan Hinz [2] and Jürgen Beyerer [1,3]

[1] Fraunhofer Center for Machine Learning, Fraunhofer Institute of Optronics, System Technologies and Image Exploitation (IOSB), 76131 Karlsruhe, Germany; jonasmohrs@gmail.com (J.M.); juergen.beyerer@iosb.fraunhofer.de (J.B.)
[2] Institute of Photogrammetry and Remote Sensing, Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany; martin.weinmann@kit.edu (M.W.); stefan.hinz@kit.edu (S.H.)
[3] Vision and Fusion Laboratory, Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany
* Correspondence: boitumelo.ruf@kit.edu

Abstract: With the emergence of low-cost robotic systems, such as unmanned aerial vehicle, the importance of embedded high-performance image processing has increased. For a long time, FPGAs were the only processing hardware that were capable of high-performance computing, while at the same time preserving a low power consumption, essential for embedded systems. However, the recently increasing availability of embedded GPU-based systems, such as the NVIDIA Jetson series, comprised of an ARM CPU and a NVIDIA Tegra GPU, allows for massively parallel embedded computing on graphics hardware. With this in mind, we propose an approach for real-time embedded stereo processing on ARM and CUDA-enabled devices, which is based on the popular and widely used Semi-Global Matching algorithm. In this, we propose an optimization of the algorithm for embedded CUDA GPUs, by using massively parallel computing, as well as using the NEON intrinsics to optimize the algorithm for vectorized SIMD processing on embedded ARM CPUs. We have evaluated our approach with different configurations on two public stereo benchmark datasets to demonstrate that they can reach an error rate as low as 3.3%. Furthermore, our experiments show that the fastest configuration of our approach reaches up to 46 FPS on VGA image resolution. Finally, in a use-case specific qualitative evaluation, we have evaluated the power consumption of our approach and deployed it on the DJI Manifold 2-G attached to a DJI Matrix 210v2 RTK unmanned aerial vehicle (UAV), demonstrating its suitability for real-time stereo processing onboard a UAV.

Keywords: embedded stereo vision; real-time stereo processing; disparity estimation; semi-global matching; GPGPU; SIMD; UAV

Citation: Ruf, B.; Mohrs, J.; Weinmann, M.; Hinz, S.; Beyerer, J. ReS^{2}tAC—UAV-Borne Real-Time SGM Stereo Optimized for Embedded ARM and CUDA Devices. *Sensors* **2021**, *21*, 3938. https://doi.org/10.3390/s21113938

Academic Editor: Jonathan Black

Received: 13 April 2021
Accepted: 28 May 2021
Published: 7 June 2021

1. Introduction

In recent years, the use and importance of unmanned aerial vehicles (UAVs) in different markets, such as aerial video and photography, precision farming, security monitoring and disaster relief, as well as 3D reconstruction and mapping has greatly increased [1–3]. And with the ongoing technological advancements, in terms of size, power and durability, the number of areas in which UAVs are used become more and more. Prototypes for delivering goods or even transporting people are already available. With the increasing use of UAVs, it is becoming evermore important that their usability and safety is ensured [4,5]. In doing so, modern UAVs are equipped with a range of sensors, including stereo vision sensors, typically used for perceiving the surrounding of the UAV to perform the tasks of obstacle detection and avoidance or 3D mapping. Compared to active sensors such as Light Detection and Ranging (LiDAR) scanners, camera systems in combination with state-of-the-art algorithms are typically more practical in performing these tasks, especially in terms of costs, weight and power consumption. Moreover, such stereo vision sensors are often already integrated in commercial off-the-shelf (COTS) UAVs.

On the other hand, while a LiDAR sensor directly provides data on the 3D geometry of the scene, using a stereo camera for the same task needs to process the stereo image data and perform a disparity/depth estimation [6]. This, in turn, requires a high-performance embedded processing on board the UAV. For a long time, so-called field-programmable gate arrays (FPGAs) were the only processing hardware that were capable of high-performance computing, while at the same time preserving a low power consumption, essential for embedded systems. In recent years, however, the availability of embedded graphic processing units (GPUs), such as the NVIDIA Tegra, allows for massively parallel embedded computing on graphics hardware, which is typically more flexible than FPGAs and less cumbersome to program. Furthermore, with the increasing use of deep learning for a wide range of applications, the importance and availability of embedded GPUs have grown even more. With the Jetson boards, comprised of an embedded ARM CPU and an embedded Tegra GPU, NVIDIA provides a suitable alternative to FPGAs for embedded high-performance computing. Especially since these systems are recently also being integrated in low-cost COTS UAVs, such as the DJI Matrix UAV in combination with the DJI Manifold or the UVify IFO-S UAV.

With this in mind, we propose an approach for real-time embedded stereo processing on ARM and CUDA-enabled devices, which is based on the well-known and widely used Semi-Global Matching (SGM) algorithm first proposed by Hirschmueller [7,8]. Our main contributions are:

- the optimization of the algorithm for embedded CUDA GPUs, such as the NVIDIA Tegra, by using massively parallel computing,
- the use of the NEON intrinsics to optimize the algorithm for vectorized SIMD processing on embedded ARM CPUs, and
- the deployment of our approach on the DJI Manifold 2-G attached to a DJI Matrix 210v2 RTK UAV and a use-case specific evaluation with respect to accuracy, processing speed and power consumption.

Even though we deployed and tested our approach for real-time processing on board a UAV, it is also suitable for other embedded systems, such as those deployed on ground-based robots or those used in advanced driver assistance systems (ADAS).

1.1. Paper Outline

This paper is structured as follows: In Section 1.2, we briefly summarize the related work on embedded stereo processing using embedded FPGA, GPU or CPU hardware and point out how our approach differs from those found in the literature. In Section 2, we first illustrate the general processing pipeline of our approach, in which we also review the general process of deriving the scene depth from a stereo image pair and provide a short review of the SGM algorithm, before illustrating in detail our optimizations for massively parallel stereo processing on CUDA-enabled GPUs and vectorized SIMD processing with NEON intrinsics ARM CPUs. We evaluate our approach on two stereo benchmark datasets with respect to accuracy, processing speed and power consumption, as well as in a use-case specific scenario. We present the results of our experiments in Section 3 and discuss our findings in Section 4, before providing a summary, concluding remarks, and a short outlook on future improvements in Section 5.

1.2. Related Work

In the following sections, we summarize the related work on embedded stereo processing. In Section 1.2.1, we will first look at studies that have deployed stereo algorithms on FPGA hardware. This is followed by an overview of the emergence of embedded GPU hardware for real-time stereo processing in Section 1.2.2. Lastly, in Section 1.2.3, we revise related work on deploying real-time stereo processing on CPU hardware, both for high-end desktop and embedded environments. We also point out how our approach differs from the related work using embedded GPU and CPU hardware for real-time stereo processing.

1.2.1. Embedded Stereo Processing on FPGAs

The use of FPGAs is key to achieve high-performance image processing with minimal power consumption, especially when relying on computationally expensive algorithms. Thus, most implementations of stereo algorithms for embedded systems, in particular of the SGM algorithm [7,8], are based on FPGA technology. First optimizations of the SGM algorithm, such as those presented in [9,10], were deployed on a PCIe-FPGA card inside a conventional PC or on a separate development kit, achieving real-time frame rates of 27 FPS and 30 FPS on low-resolution imagery, i.e., images with a size of 320 × 240 pixels and 640 × 480 pixels respectively. Due to ongoing technological advancements, the implementation of Wang et al. [11], deployed on an Altera Stratix-IV FPGA-Board, already achieved a frame rate of 67 FPS on images with a size of 1024 × 768 pixels in 2015. However, typical characteristics of embedded systems, besides the dedicated and specialized processing of a specific task, are a small form factor and the integration in larger systems or cooperative environments.

In their work, Schmid et al. [12] have deployed the implementation of Gehrig and Rabe [13] on a small quadrotor for stereo vision-based navigation achieving 14.6 FPS on a Spartan 6 FPGA. Further System-on-a-Chip (SoC) developments with respect to size and performance allowed to deploy computationally expensive algorithms on increasingly smaller systems with higher performance. Honegger et al. [14] implemented the SGM algorithm on a small SO-DIMM sized SoC equipped with a Xilinx Artix7 FPGA and reaching 60 FPS with a frame size of 753 × 480 pixels. By reducing the frame size to 320 × 240 pixels the implementation of Barry et al. [15] reached 120 FPS and was used to navigate a small and fast-flying fixed-wing UAV around obstacles.

Several recent studies [16–18] have shown that further optimizations, such as reducing the number of processing paths in the SGM optimization or increasing parallelization by splitting the input images in independent stripes, as well as the technological advancements, allow the reaching of frame rates of over 100 FPS, while at the same time increasing the accuracy of the stereo algorithm using a higher image resolution and reducing the form factor, leading to a reduced power consumption of the SoC. Yet, the use of FPGAs for real-time embedded image processing involves a cumbersome and time-consuming development, optimization and deployment process. To reduce development costs of such systems, substantial effort is done to enhance the process of high-level synthesis (HLS) and, in turn, alleviate the development of algorithms for FPGAs with more high-level languages such as C/C++ [18–20].

1.2.2. On the Emergence of Embedded Processing on GPUs

The development cycles for implementing and optimizing image processing algorithms for massively parallel processing on GPUs, on the other hand, are much shorter and thus less expensive. In addition, GPUs provide a much higher processing power, ideal for algorithms with high computational effort, such as stereo image processing. Early works [21,22] have used the rendering pipeline of OpenGL to deploy the SGM stereo algorithm on graphics hardware and reached frame rates of up to 8 FPS and 4 FPS on VGA image resolution, respectively. With the introduction of the CUDA-API in 2007, the development costs for general-purpose computation on a GPU (GPGPU) have dropped even more. And so, a lot of implementations of the SGM algorithm for real-time stereo processing on hardware without embedded constraints are optimized and deployed on graphics hardware [23–25]. Hernandez-Juarez et al. [25] show that with increasing computational power, the use of GPGPU on modern, high-performing graphics hardware, such as the NVIDIA Titan X, allows the reaching of frame rates of up to 237 FPS on VGA image resolution with the conventional use of eight optimization paths inside the SGM optimization. Even higher frame rates of up to 475 FPS and 886 FPS are possible, if the number of optimization paths are reduced to four or two, respectively.

Although GPUs provide great computational performance, a major drawback is given by their high power consumption. The deployment of the SGM algorithm on a high-

end GPU only achieves 1.90 FPS/W on VGA image resolution [25], while a comparable configuration of the algorithm deployed on a state-of-the-art embedded FPGA achieves 15 FPS/W on a larger image resolution [18]. However, due to the increasing importance of deep learning algorithms for robotic applications, more and more embedded CPU-GPU-based SoCs are released and integrated into robotic systems. Recently, such SoCs with embedded GPUs have even been integrated onto COTS UAVs, and thus been made available for the mainstream user. In their work, Hernandez-Juarez et al. [25] have also deployed their implementation on the NVIDIA Jetson TX1, encapsulating the embedded Tegra X1 GPU, reaching 42 FPS (4.19 FPS/W) on VGA resolution and a four path SGM optimization. Chang et al. [26] further optimized the computation of the normalized cross-correlation (NCC) matching cost for the use on GPUs and deployed their SGM-based stereo algorithm on the NVIDIA Jetson TX2 reaching 28 FPS on images with a size of 1242×375 pixels.

In our work, we have also optimized the SGM algorithm for real-time processing on CUDA devices and deployed it on the NVIDIA Jetson TX2 and the more powerful NVIDIA Jetson Xavier AGX. In this, we evaluate the performance of different configurations and optimization strategies with respect to performance and power consumption. The computationally most expensive part of the SGM algorithm is the aggregation along the different scanlines. At the same time, due to the nature of dynamic programming, this is also the part which can by parallelized most effectively, since the computation of each scanline can be done fully independently, without the need for synchronization. In terms of GPGPU, this is typically done by instantiating one thread on the graphics hardware for each scanline, resulting in a massively parallel processing SGM path aggregation. We have adopted this approach as described in Section 2.2.

1.2.3. Are Embedded CPUs Suitable for Stereo Processing?

In contrast to FPGAs and GPUs, which are designed to be less flexible and yet very powerful in processing specific tasks on a large amount of data, CPUs are designed to do more general and versatile processing, needed to allow computers to instantly react to new sensor input. Even though they have much higher clock frequencies, CPUs are often not capable of keeping up with the performance achieved by their more specialized counterparts, due to their small number of cores and, in turn, limited ability of parallelization.

One of the first deployments of the SGM algorithm on a conventional CPU was done by Gehrig and Rabe [13]. They have implemented several different parallelization techniques, among others splitting the eight-path optimization scheme into two independent scans. With this, they achieve 14 FPS on images with a size of 640×320 pixels, but they only considered a range of 16 disparities. They have deployed their algorithm on an Intel Core i7 with four cores and a clock frequency of 3.3 GHz. A few years later, Spangenberg et al. [27] have achieved 16 FPS on VGA resolution and 128 disparities, also running their implementation on a conventional Intel Core i7 with four cores. Apart from several algorithmic optimizations, e.g., disparity space compression and striped computation, they have parallelized the processing by using Single-Instruction-Multiple-Data (SIMD) vectorization with the SSE instruction set from Intel, combined with multi-threading.

The work of Arndt et al. [28] is one of the first to deploy an implementation of the SGM algorithm on an embedded CPU, namely the Freescale P4080, reaching a frame rate of only 0.5 FPS on VGA image resolution. When considering that this was done around the same time as the works of Gehrig and Rabe [13] and Spangenberg et al. [27], it illustrates the gap between conventional and embedded CPUs in terms of performance. However, the embedded CPU technology has also evolved and gained performance. Most of the modern high-end embedded SoCs typically consist of an ARM CPU and a FPGA or a GPU, for example the Xilinx Ultrascale series or the NVIDIA Jetson series. Rahnama et al. [29] implemented the ELAS stereo algorithm [30] on a Xilinx ZC706 SoC made up of an ARM CPU and a FPGA. In this, they have deployed the computationally most expensive stages of the algorithm onto the FPGA (if possible) and used the ARM CPU to process the stages

with unpredictable memory access patterns. Saidi et al. [31] accelerated a simple stereo algorithm on an ARM CPU achieving frame rates of up to 59 FPS on an image resolution of 320 × 240 pixels. In this, they have parallelized the algorithm using multi-threading as well as SIMD vectorization on ARM with the NEON instruction set [32].

Just as the dynamic programming in the SGM path aggregation is well suited for massively parallel computing on graphics hardware, its computation can also be effectively vectorized by SIMD processing as Spangenberg et al. [27] have shown. With this in mind, we want to investigate the ability of embedded CPUs in performing high-accuracy stereo processing in real time based on the SGM algorithm. To the best of our knowledge, our work is the first to implement and optimize the Semi-Global Matching stereo algorithm on an embedded ARM CPU, leveraging multi-threading parallelization and SIMD vectorization with NEON intrinsics. We have summarized a relevant excerpt of the related work on real-time SGM stereo processing on FPGA, GPU and CPU hardware in Table 1.

Table 1. Overview of numerous studies that have developed and deployed an SGM-based stereo algorithm on different hardware architectures. The highest frame rates, with respect to the corresponding power consumption, are achieved by FPGA-based implementation. ∗: Power consumption stated for the whole system, e.g., including image capture. †: Studies state measurements for different configurations; however, the configuration listed provides a good trade-off between image resolution and frame rate.

Reference	HW Device	Embedded SoC	Resolution	Disp. Range	FPS	Power
Gehrig et al. [9]	FPGA		320 × 240	64	27	<3 W
Banz et al. [10]	FPGA		640 × 480	128	30	n/a
Honegger et al. [14]	FPGA	✓	753 × 480	32	60	<5 W
Wang et al. [11] †	FPGA		1024 × 768	96	67	n/a
Barry et al. [15]	FPGA	✓	320 × 240	32	120	<5 W ∗
Hofmann et al. [16] †	FPGA	✓	640 × 480	64	140	n/a
Ruf et al. [19]	FPGA	✓	640 × 360	64	29	n/a
Rahnama et al. [17] †	FPGA	✓	640 × 480	128	109	<3 W
Zhao et al. [18] †	FPGA	✓	1242 × 374	128	161	6.6 W
Banz et al. [23] †	GPU		1024 × 768	128	25	n/a
Michael et al. [24]	GPU		640 × 480	64	11.7	n/a
Hernandez-Juarez et al. [25] †	GPU	✓	640 × 480	128	42	<10 W
ReS^{2}tAC-CUDA (ours)	GPU	✓	640 × 480	128	24	~20 W ∗
Gehrig and Rabe [13] †	CPU		640 × 320	16	14	n/a
Arndt et al. [28] †	CPU	✓	640 × 480	64	0.5	n/a
Spangenberg et al. [27] †	CPU		640 × 480	128	16	n/a
ReS^{2}tAC-NEON (ours)	CPU	✓	640 × 480	128	7.2	~18 W ∗

2. Materials and Methods

In the following sections, we first give a general overview of the processing pipeline of our approach (Section 2.1), in which we also review the general process of deriving the scene depth from a stereo image pair and provide a short review of the SGM algorithm. This is followed by detailed descriptions on our optimizations for massively parallel stereo processing on CUDA-enabled GPUs (Section 2.2) and vectorized SIMD processing with NEON intrinsics ARM CPUs (Section 2.3).

2.1. Processing Pipeline for Real-Time Dense Disparity Estimation

In their work, Scharstein and Szeliski [33] have studied and categorized several stereo algorithms for dense disparity or depth estimation based on their processing steps. From their observations they have derived a general processing pipeline which provides a basic blueprint for most modern stereo algorithms, as well as for our approach proposed in this work, as illustrated in Figure 1. In this, the estimation of dense disparity maps can be subdivided into three subsequent steps, which are made of smaller building blocks. In the following, we will look at each individual processing step, give a short overview of its task,

and subsequently provide a detailed description on how we have instantiated it in our proposed approach.

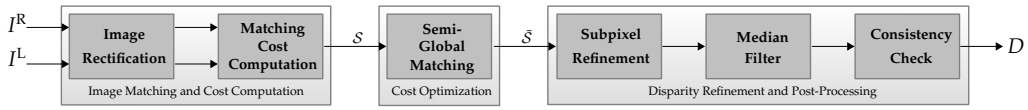


Figure 1. Processing pipeline for real-time dense disparity estimation consisting of three subsequent steps, which in turn are made of smaller building blocks.

2.1.1. Dense Image Matching and Cost Computation

Finding two corresponding image points that depict the same scene point in both images of a stereo camera setup (Appendix A), means to match the pixels of the reference image, typically the image of the left camera, against each pixel in the matching image within a certain disparity range $d \in \Gamma = [d_{\min}, d_{\max}]$. In this, the similarity between these the two pixels is modeled by a similarity measure from which a cost function Φ can be deduced, which typically is at its minimum if both pixels coincide. This so-called matching cost between a pixel p in the left image and a corresponding pixel q, located according to a disparity shift d in the right image, is stored in the three-dimensional cost volume $\mathcal{S}$:

$$\mathcal{S}(\mathrm{p}, d) = \Phi\Big(I^{\mathrm{L}}(p_x, p_y), I^{\mathrm{R}}(p_x - d, p_y)\Big). \tag{1}$$

Thus, the objective in finding two corresponding image points is to minimize the matching cost computed by Φ.

When relying on distinctive image features such as SIFT [34], SURF [35] or ORB [36] features, a unique matching between two image points can be found. However, such image features can only be calculated in descriptive image regions, resulting in a very sparse correspondence field. Thus, in the case of dense image matching Equation (1) is evaluated for each pixel in the reference image, computing a pixel-wise matching cost s according to Φ, which indicates the similarity between the pixel pair. This cost is then stored inside a three-dimensional cost volume $\mathcal{S}$ from which the disparity map is later extracted. In our work, we have implemented the two most commonly used similarity measures for real-time dense image matching, namely the Hamming distance of the non-parametric census transform (CT) [37] and the normalized cross-correlation (NCC). Since the Hamming distance of the CT is minimal when both image patches are most similar, it can directly be used as the matching cost s_{CT}. The NCC, however, is equal to 1 when both image patches are equal. Thus, the NCC is inverted and truncated before being evaluated as matching cost: $s_{\mathrm{NCC}} = 1 - \max(0, \Phi_{\mathrm{NCC}})$.

Since the disparity is only evaluated along the same pixel row (Equation (1)), it is assumed that the input images I^{L} and I^{R} are rectified prior to the process of image matching. Similar to the way described by Ruf et al. [19], we use a standard calibration routine implemented in the OpenCV library [38] to calculate two rectification maps, which allow efficient resampling of the input images such that the epipolar lines lie horizontally on the image rows.

2.1.2. Cost Optimization and Disparity Computation

Given the previously computed three-dimensional cost volume $\mathcal{S}$, the next step consists of extracting a plausible disparity map D. Since each voxel of $\mathcal{S}$ holds the matching cost of a particular pixel p of the reference image, associated with a certain disparity d within the studied disparity range, a straightforward approach to compute a disparity map from the cost volume is to take the *winner-takes-it-all (WTA)* solution for all pixels inside the reference image:

$$D(\mathrm{p}) = \underset{d \in \Gamma}{\arg\min}\, \mathcal{S}(\mathrm{p}, d). \tag{2}$$

However, due to the limited descriptiveness of the cost functions and resulting ambiguities in the cost volume, this would lead to a noisy and unsuitable disparity map. Thus, it is important to perform a cost optimization and in turn regularize the cost volume.

Scharstein and Szeliski [33] have categorized the stereo methods according to their cost optimization strategies into local and global methods. Although the first group of algorithms only optimize the cost volume in a locally confined window and make implicit smoothness assumptions, global methods explicitly state their regularization scheme and perform an optimization within the whole image domain. Global methods will thus produce more accurate disparity maps compared to those estimated by local methods. Yet, at the same time, the use of global methods for embedded stereo processing is not feasible due to their complexity.

In their taxonomy, Scharstein and Szeliski [33] additionally propose a third group of algorithms, which alleviate dynamic programming to compute a disparity map. Algorithms in this group usually state an explicit smoothness assumption and approximate a global optimization scheme, which is why this group can be considered to be a subgroup of the global methods. The most prominent algorithm, especially for real-time embedded processing, is the so-called Semi-Global Matching (SGM) algorithm [7,8]. In this, the optimization scheme is formulated as a Markov Random Field (MRF) and the minimization of the energy function with its explicit smoothness assumption is approximated by aggregating the matching costs along several concentric paths for each pixel p within the image domain:

$$L_{\mathrm{r}}(\mathrm{p}, d) = \mathcal{S}(\mathrm{p}, d) + \min_{d' \in \Gamma} \big(L_{\mathrm{r}}(\mathrm{p} - \mathrm{r}, d') + \mathcal{V}(d, d') \big), \text{ with}$$
$$\mathcal{V}(d, d') = \begin{cases} 0, \text{ if } d = d' \\ P_1, \text{ if } |d - d'| = 1 \\ P_2, \text{ if } |d - d'| > 1. \end{cases} \tag{3}$$

For each pixel p and disparity d inside the cost volume $\mathcal{S}$, the matching costs are recursively aggregated into L_{r}, while moving along the path with the direction r. Within the smoothness term $\mathcal{V}(d, d')$ the matching costs of the previously considered pixel, with respect to all evaluated disparities d', are penalized according to the disparity difference between d and d'. Finally, for each pixel, all path costs are summed up and stored inside an aggregated cost volume $\bar{\mathcal{S}}$:

$$\bar{\mathcal{S}}(\mathrm{p}, d) = \sum_{\mathrm{r}} L_{\mathrm{r}}(\mathrm{p}, d), \tag{4}$$

before extracting the WTA disparity map according to Equation (2) from the same.

The use of dynamic programming, i.e., breaking down the minimization problem of the energy function into the aggregation of independent one-dimensional paths, makes the SGM approach well suited for massively parallel computing and vector processing, and in turn for embedded processing. At the same time, many studies have shown that the results of the SGM algorithm are very accurate, making it one of the most widely used algorithms for real-time and accurate stereo processing. In our work, we have parallelized and optimized the SGM algorithm to run in real time with SIMD vector processing on embedded ARM CPUs as well as CUDA-enabled GPUs.

2.1.3. Post-Processing

There are several post-processing steps, i.e., filtering, regularization and further optimizations, which can be applied to the initial disparity map in the final stage of the processing pipeline. In our pipeline, we have implemented a subpixel disparity refinement, a left–right consistency check for occlusion detection, as well as a final median filter.

Subpixel Disparity Refinement

The initial disparity map computed by the SGM optimization is made up of discrete disparity values, which is sufficient for a diversity of robotic applications such as perception of the surrounding and obstacle detection. However, when the aim is to accurately reconstruct the scene, it is important to also account for slanted surfaces and thus incorporate a subpixel refinement of the disparity. A simple and yet effective way to implement such a refinement is to use the minimum matching cost for each pixel, i.e., the matching cost corresponding to the WTA disparity $\hat{d}$, as well as the matching costs of the two neighboring disparities in front and behind of $\hat{d}$, and fit a parabola through these three matching costs. The location of the minimum of this parabola with respect to $\hat{d}$ is then considered to be the subpixel refinement and added to $\hat{d}$. This optimization has only a minor computational overhead and is thus well suited for real-time processing. However, it needs to work with floating point arithmetic which might be of restriction to some embedded hardware.

Occlusion Detection by Left–Right Consistency Check

A typical approach to detect and filter pixels in occluded areas is the left–right consistency check (cf. Appendix B). This, however, requires the computation of a second disparity map D^{R}, which corresponds to the right image of the stereo pair. A straightforward approach to compute D^{R} would be to swap and horizontally flip the input images and repeat the image matching, cost optimization and disparity computation as described above. This, however, would mean to execute the first and computationally most expensive steps of the processing pipeline twice for each image pair. Yet, the computation of D^{R} can be efficiently approximated by reusing the aggregated cost volume $\bar{\mathcal{S}}$ from the cost optimization step:

$$D^{\mathrm{R}}(\mathrm{p}) = \arg\min_{d \in \Gamma} \bar{\mathcal{S}}((p_x + d, p_y), d). \tag{5}$$

In this work, we employ the approximated computation of D^{R} for real-time processing and evaluate how it performs compared to computing the right disparity map from scratch.

Median Filter

For a final outlier removal, we employ a 3×3 median filtering for all remaining pixels with valid disparities. A median filtering requires a sorting of all disparities within the local window, which is especially cumbersome when optimizing for vector processing on a CPU. Thus, we are using sorting networks to efficiently perform a bubble sort when running the algorithm on the ARM CPU (cf. Section 2.3.4). We chose a confined neighborhood size of 3×3 pixels because of two reasons: First, to not introduce too much smoothing or object fattening in the resulting disparity map. And secondly, to keep the computational complexity in the process of sorting the disparity values low.

2.2. Real-Time Processing by Massively Parallel Computing on CUDA-Enabled GPUs

As described in Appendix C, the CUDA-API allows for massively parallel general-purpose computation on a GPU (GPGPU) on NVIDIA GPUs. Thus, we have implemented each step of the stereo algorithm in separate CUDA kernels to optimize the stereo algorithm for embedded NVIDIA GPUs. Since each kernel execution is aimed to achieve a high use of the GPU, we refrained from a parallel execution of the CUDA kernel methods with CUDA streams. In the following, we provide a detailed description on how we have optimized and instantiated each step of the algorithm for an efficient GPGPU.

2.2.1. Matching Cost Computation

As already mentioned in Section 2.1.1, we have implemented two different matching cost functions, namely the Hamming distance of the census transform (CT) as well as an inverted and truncated version of the normalized cross-correlation (NCC). A detailed description on the implementation and the optimization of these two cost functions for execution on a GPU with CUDA is given in the following.

Calculating the Census Transformation and Its Hamming Distance

For the parallel calculation of the CT on the GPU, we assign different image regions to each instantiation of the corresponding CUDA kernel. Before the actual CT is calculated, each kernel instantiation copies the pixel data of the considered image region into shared memory to achieve a higher access speed. The computation of the CT is then performed in parallel by each thread of the thread-block for a specific pixel in the assigned image region. To account for pixels at the image border, where a part of the neighborhood lies outside of the image, a zero-valued margin with the size of the neighborhood radius is assigned to the image for the calculation of the CT. Furthermore, we separated the calculation of the Hamming distance from the calculation of the CT, since the two kernel methods of these two steps are instantiated with different parameters. For the parallelization on the GPU, we have implemented a CT with a neighborhood size of 5×5 pixels and 9×7 pixels, the latter being the largest neighborhood that fits into a 64 bit integer. The choice of the former neighborhood size is justified by the limitations of the optimized implementation for the CPU (cf. Section 2.3.1). An overview of our implementation is illustrated by Figure 2 (top).

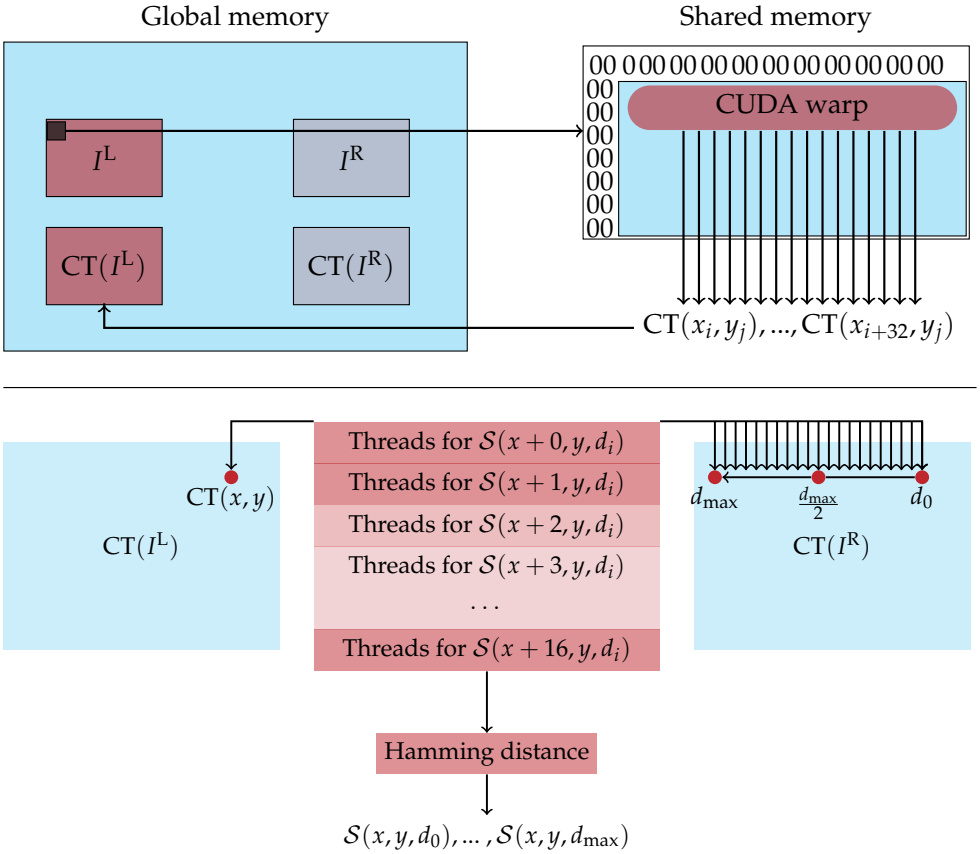


Figure 2. **Top:** To calculate the census transform by a CUDA warp for a specific image region, the image data of this region is first copied from the global memory to the shared memory, the latter having higher access speeds. In the second step, each thread of the CUDA warp calculates the census transform (CT) for one pixel inside this region. **Bottom:** The threads of a CUDA warp calculate the Hamming distance for 16 disparities simultaneously.

In the calculation of the Hamming distance, the reference and matching image are divided into stripes and each stripe is assigned to different CUDA warps. Each thread of the CUDA warps then calculates the Hamming distance from the corresponding census descriptors at a certain pixel position and disparity. We have chosen the dimensions of the CUDA warps in such a way that for 16 different pixels half the disparities can be calculated simultaneously (Figure 2 (bottom)). In this, the census descriptor of the reference pixels is

first loaded by all threads of a thread-block into the shared memory. Then all threads load the census descriptors of the matching pixel given a certain disparity. This means that the 32 threads of a thread-block load the census descriptors $CT(x-i,y),\ldots,CT(x-i-31,y)$ from the matching image into the shared memory. This is repeated until for all disparities $[0,d_{\max}]$ the census descriptors of the matching image are loaded into the shared memory. Given all the census descriptors, the threads of a thread-block compute the Hamming distance simultaneously for different pixels and store the result at the corresponding position in the cost volume. Since the matching cost of the different disparities for one pixel lie directly next to each other in the cost volume, the dimension of the CUDA warp is chosen in such a way that the memory access from the GPU can be pooled together.

Inverted and Truncated Version of the Normalized Cross-Correlation

Using the NCC as a cost function is computationally more expensive than relying on the Hamming distance of the CT. Although the computation of the CT and the subsequent Hamming distance only requires some comparative and bit-level operations, the computation of the NCC needs the calculation of a mean and variance of the two input patches. Since the mean and the variance of all possible patches inside an image can be precomputed and then reused, we divide the calculation of the NCC and the process of image matching into two separate stages, similar to the calculation of the CT and the Hamming distance.

In the first step, we calculate the mean and variance for all patches in the left and right input image. In this, we instantiate a kernel with the same configuration as when calculating the CT, iterating over all pixels in the left and right image, and compute the necessary data for a patch of a given size, centered around the current pixel. Similar to the process of computing the CT, we thus first enhance the input images with the independent patch information, storing the patch-mean and patch-variance, together with the pixel value of the center pixel, in a special struct for each pixel of the input image. Just as when calculating the Hamming distance of the census transform, we then use the pixel and the patch data to perform the image matching based on the inverted and truncated normalized cross-correlation and fill the resulting cost volume in the second stage. Again, we instantiate a kernel with the same parameters, as when doing the image matching with the Hamming distance. We have implemented the NCC for a patch size of 5×5 pixels and 9×9 pixels.

2.2.2. Semi-Global Matching Optimization

The calculation of the eight different SGM path costs is done sequentially on CUDA hardware. The parallelization of the cost aggregation on one path direction is realized on two different levels. First, each CUDA block calculates the costs for 16 different lines along one path direction (Figure 3). If a diagonal line reaches the image border, the values are reset and the calculation is resumed on the other side of the image. This ensures that all calculations of one path direction takes the same time. Additionally, for each image point, the costs for the disparities $d_0,\ldots,(\frac{d_{\max}}{2}-1)$ and $\frac{d_{\max}}{2},\ldots,d_{\max}$ are calculated in parallel by two iterations. This ensures that all threads within one warp access a contiguous area in the memory, allowing the memory transactions to be more efficient. However, this requires a synchronization of the threads within a CUDA warp after the costs for all disparities have been calculated, to find the minimum path cost, which is necessary for further processing. To find the minimum of the aggregated costs, we use the map reduce method as illustrated in Appendix D. After the calculation and aggregation of the different SGM path costs, the WTA disparity with the minimum aggregated costs is to be found. This is done by assigning a specific image region to each CUDA warp and again using the map reduce method mentioned above.

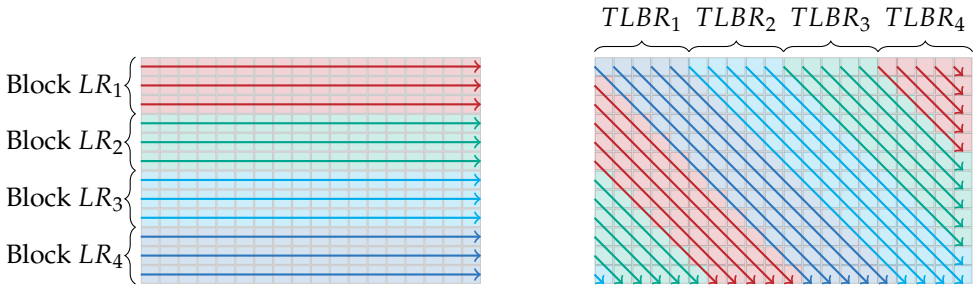


Figure 3. Multiple CUDA blocks processing the SGM path aggregation. Each block calculates 16 different lines along one path direction.

2.2.3. Consistency Check

The key aspect in the consistency check is the calculation of the approximated disparity map D^{R}, corresponding to the matching image. This is approximated from the calculated aggregated cost volume of the reference image $\bar{\mathcal{S}}$. In this, each entry of D^{R} is calculated according to Equation (5). The difficulty that arises in this process is the access of non-adjacent areas in $\bar{\mathcal{S}}$, as illustrated in Figure 4. Between all entries of the aggregated cost volume $\bar{\mathcal{S}}$ that are to be used for the calculation of D^{R}, always lie $d_{max} + 1$ entries, which are of no interest.

In the GPU implementation for the consistency check, each instantiation of a CUDA kernel computes a specified region in the approximated disparity map D^{R}. In this, the data of the aggregated cost volume is first copied into shared memory for quicker access. Since the data does not lie next to each other, the access to the global memory by the different CUDA threads cannot be pooled together. When all data are available in the shared memory, we again use the map reduce method to find the minimum. After the disparity map D^{R} for the matching image is approximated, each thread performs the consistency check according to Equation (A1) for a specific pixel in the final disparity map D^{L}.

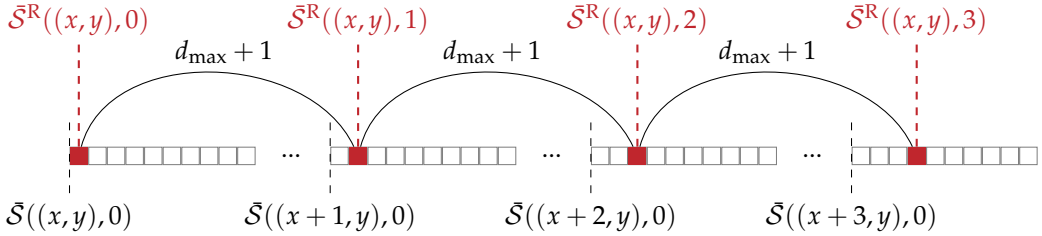


Figure 4. In the consistency check, an additional disparity map, which corresponds to the matching image, is approximated from the aggregated cost volume $\bar{\mathcal{S}}$. The required entries are not situated directly next to each other, which hinders an efficient memory access.

2.2.4. Median Filter

The execution of the median filter on the GPU is straightforward. To each CUDA thread-block a region in the final disparity map is assigned for which the filter is to be processed. In this, the necessary data are first copied to the shared memory. Pixels that lie outside the image achieve a disparity value of 0xFFFF assigned, making them irrelevant for the filtering. With all the data in the shared memory, each thread of the thread-block calculates the median for a pixel. In this, the first five iterations of the bubble sort algorithm are performed to sort the values in the 3×3 pixels neighborhood. After the fifth iteration, the five highest values are correctly sorted, and the median can be extracted.

2.3. Vectorized SIMD Processing with NEON Intrinsic Set on ARM CPUs

To efficiently deploy the real-time processing pipeline for the estimation of dense disparity maps on an embedded CPU, such as the 8-core ARMv8.2 on the NVIDIA Jetson Xavier AGX, we use two strategies of parallelization as illustrated in Figure 5, namely:

1. a thread-level parallelization, and
2. a vectorized data processing with the Single-Instruction-Multiple-Data (SIMD) NEON intrinsics.

Our implementation uses eight concurrent threads to efficiently use the available CPU cores. In each step of the processing pipeline, in exception to the SGM optimization step, each thread operates on a different image stripe, thus operating isolated and independently from the other threads. At two locations in the processing pipeline of the stereo algorithm, i.e., before and after the SGM optimization, the concurrent threads need to be synchronized, since the SGM optimization relies on a different thread barrier than the other steps of the pipeline. In the other steps of the pipeline, the threads do not require any synchronization and can thus be processed fully concurrently. As part of our second parallelization strategy, each thread uses the ARM NEON instruction set [32] to perform a vectorized SIMD processing on the vector-processors of the CPU (cf. Appendix E).

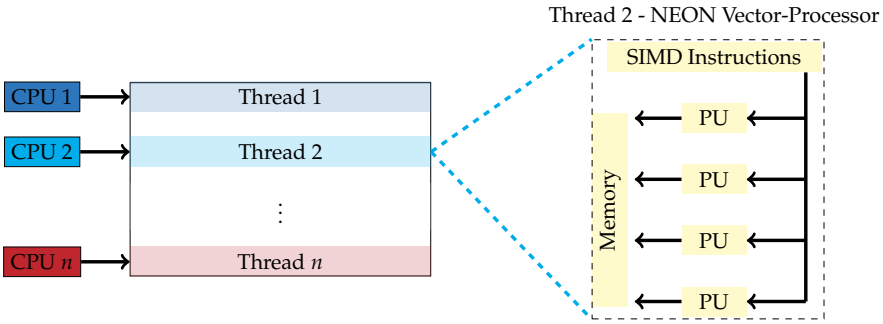


Figure 5. In the optimization of the SGM stereo algorithm for the execution on embedded CPUs, two different parallelization strategies are used. The implementation uses multiple threads to evenly distribute the processing on the available CPU cores. Each thread uses the AMD NEON instruction set to perform a vectorized SIMD processing using the NEON processing units (PU).

In the following sections, we will discuss how we have optimized each step of the processing pipeline for execution on an ARMv8 CPU, using thread parallelism and SIMD processing. Since the use of the NCC as a cost function for the image matching is computationally more expensive, and the frame rates achieved using the CPU are anyway much lower than those achieved on the GPU, we refrained from implementing the NCC for a vectorized SIMD processing on the CPU.

2.3.1. Calculating the Census Transformation and Its Hamming Distance

As illustrated in Figure 6, we can calculate the census transform (CT) in each thread for 16 pixels simultaneously, using the SIMD vector-registers. In this, we load the 16 reference pixels with eight bits each into one vector-register. In addition, we load the corresponding 24×16 neighbor pixels into one vector-register each, resulting in a total use of 25 vector-registers with 16 lanes. We process the full image by sliding the illustrated window from left to right and top to bottom over the image. In doing so, there is a good chance that the image data, which is needed by the next iteration, is already cached. In the calculation of the CT, we omit the comparison of the reference pixel with itself. This allows us to represent the 24 bits of the resulting CT bitstring by three bytes and thus store the census descriptors of all 16 pixels in three vector-registers. This is also why we only consider a 5×5 pixels neighborhood in the optimized implementation of the CT for the ARM CPU. To also calculate the CT at the image border, where a part of the neighborhood lies outside of the image, we would need to introduce a conditional statement, which is not recommended when using SIMD vectorization. Thus, we only calculate the CT up to two pixels with respect to the image border, reducing the image size by four pixels in each dimension for the subsequent processing.

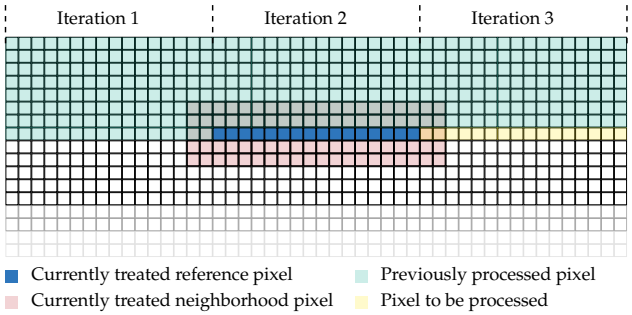


Figure 6. For the optimized CPU implementation, the census transform is processed for multiple pixels simultaneously by using the NEON vector processing units. In this, a sliding window is used, which processes the image data from left to right and from top to bottom.

To calculate the Hamming distance between two census descriptors, namely the one from the pixel of the reference image and the corresponding pixel in the matching image, we apply the XOR operator and count how many bits are set to 1 in the final output. To count the number of bits which are set, the NEON instruction set offers a population count (VCNT) which can be applied to each vector-lane. As illustrated in Figure 7, the resulting matching cost, i.e., the Hamming distance of the CT, is stored to a three-dimensional cost volume. By using the SIMD instructions and the 32 vector-register, a maximum of 64 matching costs can be calculated simultaneously in each thread. Thus, each thread processes 16 disparities and four lines simultaneously in one iteration.

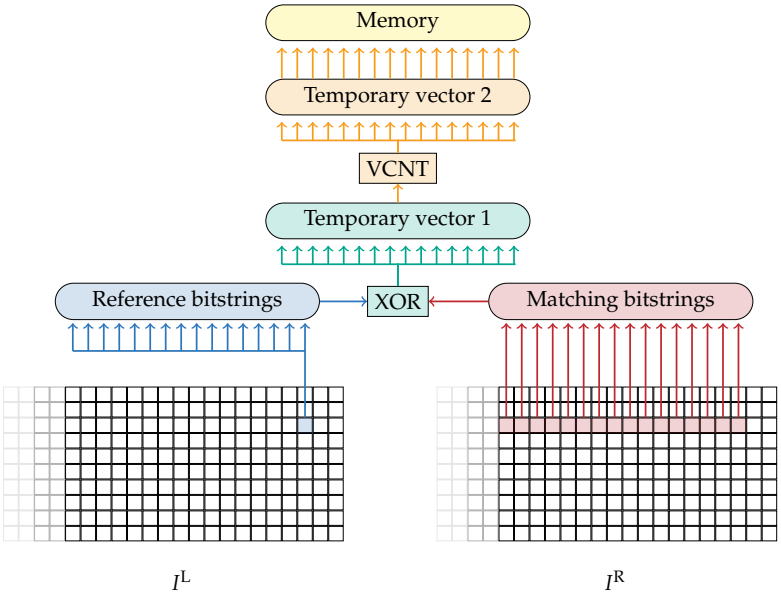


Figure 7. Overview of the calculation of the Hamming distance with SIMD intrinsics. Each census descriptor is loaded from I^L and I^R in separate vector-registers, on which a XOR operation is applied. The number of set bits inside a vector-register is counted using the NEON hardware instruction VCNT (Vector Count Set Bits).

In the case the currently calculated disparity is bigger than the x-coordinate of the reference pixel, the corresponding matching pixel lies outside the image. To efficiently handle this case, we additionally store the disparity for which the matching costs are currently being calculated as well as the x-coordinate of the currently processed pixel in two additional vector-registers. In each iteration, both of the above-mentioned registers are compared against each other, and the result is stored in a third register. If the disparity is greater than the x-coordinate of the pixel, all bits inside the vector-lanes will be set. Finally, if we apply an OR operation between the register with the matching cost and the register

with the comparison result, the matching cost of each disparity that spans over the image boundary will be set to 0xFF and thus will not contribute in the subsequent search for the optimum.

2.3.2. Semi-Global Matching Optimization

The optimized implementation of the SGM algorithm can be divided into two separate steps. First, each thread calculates the SGM path costs for each path that is assigned to it according to Equation (3). As illustrated in Figure 8, the four vector-registers are first filled with the results of the previous iteration, so that the vector-register $L_\mathrm{r}(\mathrm{p}-\mathrm{r},d)$ will hold the previous path costs at the same disparity level, the vector-registers $L_\mathrm{r}(\mathrm{p}-\mathrm{r},d-1)$ and $L_\mathrm{r}(\mathrm{p}-\mathrm{r},d+1)$ will hold the previous path costs at the disparity level ± 1, and the vector-register $\min_d L_\mathrm{r}(\mathrm{p}-\mathrm{r},d)$ will hold the minimum path costs over all disparity levels at the considered pixel. According to Equation (3), the current matching costs from the cost volume as well as the penalties are added to the different vector-registers. Again, the NEON instruction set provides a method to achieve the minimum from the four vector-registers, namely $\mathrm{MINIMUM}_\mathrm{shuffle}$. The result is stored in the allocated memory and is compared to the path costs of the other disparities to achieve the minimum path cost for the next iteration.

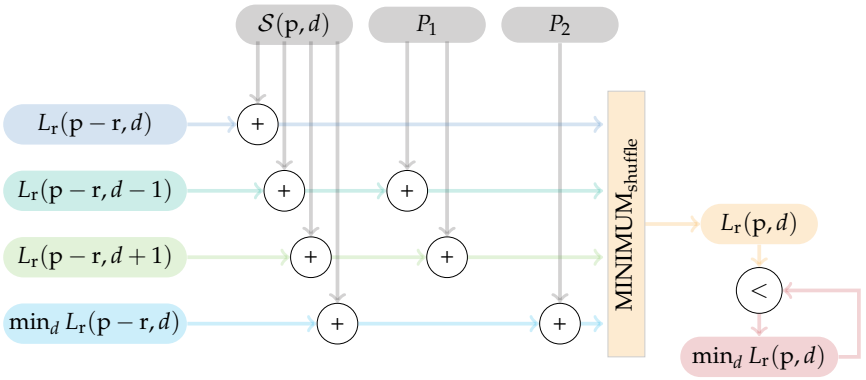
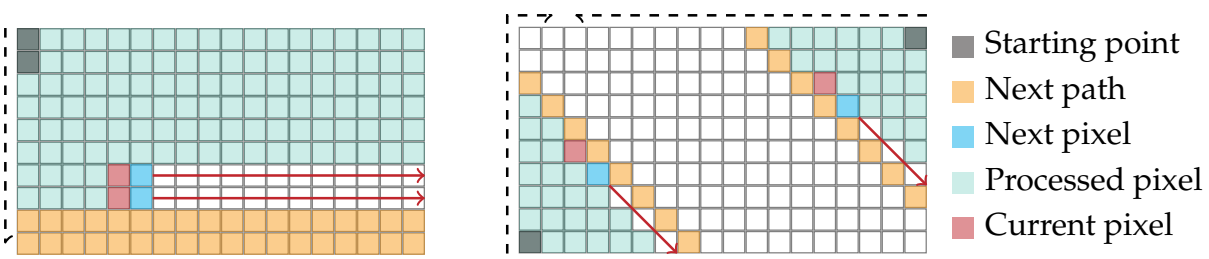


Figure 8. Schematic overview of the implementation of the SGM aggregation at a single pixel on a path. All components of the SGM path aggregation (cf. Equation (3)) are calculated simultaneously. A final minimum operation will yield the result which is stored in the aggregated cost volume.

In each thread, the costs for the disparities of two pixels are calculated simultaneously. In this, the implementation of the horizontal and vertical paths (L_LR, L_RL, L_TB, L_BT) differ from the implementation of the diagonal paths (L_TLBR, L_TRBL, L_BLTR, L_BRTL). On the straight paths, each thread processes two pixels on two neighboring rows or columns (Figure 9a). However, due to the different lengths of the diagonal paths, this cannot be applied to the processing of the same. Instead, on the diagonal paths, each thread processes two pixels lying on opposite sides of the image. This is illustrated in Figure 9b.



(a) Straight paths **(b)** Diagonal paths

Figure 9. Different traversal strategies in the aggregation of the SGM path costs.

In the second step of the SGM aggregation, each thread sums up all SGM path costs and finds the disparity with the minimum cost, i.e., the WTA cost, for each image stripe assigned to it. The different path costs are first copied into different vector-registers and then summed up. When summing up, each thread additionally stores the currently processed disparity as well as the minimum aggregated cost and the corresponding WTA disparity in additional vector-registers. Afterwards, the aggregated costs are compared to the minimum costs which are updated if necessary. If the minimum costs are updated, the corresponding WTA disparity is updated accordingly. The final aggregated path costs are stored into an aggregated cost volume $\bar{S}$, which is needed for the subsequent consistency check, while the final WTA disparity is written into the disparity map.

2.3.3. Consistency Check

In the optimization of the consistency check for the CPU, we encounter the same difficulty as in the implementation of the approximated consistency check for the GPU, namely that the required data from the aggregated cost volume does not lie physically next to each other (cf. Figure 4). This makes it inefficient to load the data into vector-registers first and then process it with NEON intrinsics. There are two ways to solve this problem: The first possibility, which is proposed by Spangenberg et al. [27], is to first transform the aggregated cost volume $\bar{S}$ into a temporary volume in such a way that the data which is required to compute the approximated disparity map D^{R} will physically lie next to each other in memory. Afterwards, the vector-registers can be filled with the data and the approximated disparity map D^{R}, corresponding to the matching image, can be calculated efficiently with SIMD instructions. The second option, which we chose, is to refrain from using SIMD instructions for the consistency check. In our implementation of the consistency check, we only use a thread-level parallelization in which each thread is processing a different part of the cost volume. This saves us the need to rearrange the cost volume necessary to use SIMD instructions.

2.3.4. Median Filter

In our pipeline, we deploy a final median filter after the consistency check to remove any small outliers that remain in the disparity map D. This requires a sorting of all disparity values within the local neighborhood. We use the concept of sorting networks (cf. Appendix F), which relies on wires and comparators, to allow for a parallel sorting with SIMD intrinsics. In the implementation for the vectorized processing of the 3×3 median filter, the wires of the sorting network are realized using the vector-lanes of the vector-register. In this, the wires of the network are distributed among different vector-registers, using one vector-lane from each register. Thus, for the implementation of one sorting network for the median filter with nine wires, nine different vector-lanes distributed over nine different vector-registers are used (Figure 10).

The comparators of the sorting network are implemented using two comparison instructions of the NEON instruction set that compare each vector-lane of two vector-registers and store the minimum or maximum in a third register. Thus, for each vector-lane we first extract the minimum and maximum value from the two source vector-registers. Then, the minimum is copied to the register which represents the upper lane of the sorting network, while the maximum is copied to the register which represents the lower lane. By this vectorized parallelization the median filter is computed for 16 pixels simultaneously. Inherent to the nature of the BubbleSort algorithm, it is only necessary to calculate the first five iterations to achieve the median of a 3×3 pixels neighborhood.

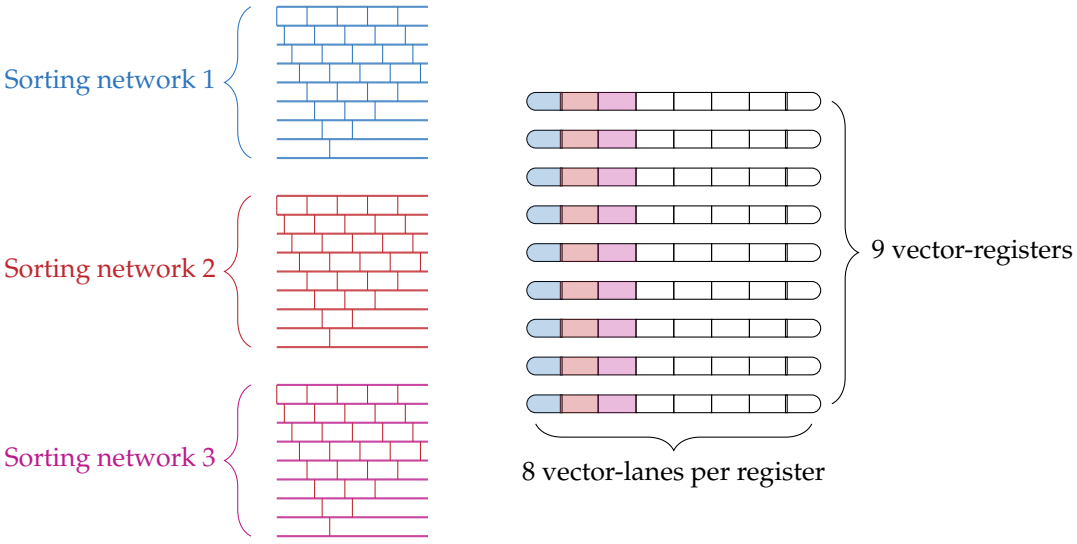


Figure 10. Implementation of sorting networks using vectorized SIMD processing. The nine wires of a sorting network are mapped to vector-lanes of nine different vector-registers.

3. Results

The first part of our experiments (Section 3.1) comprises a quantitative evaluation of the accuracy and performance of our optimized implementations on the KITTI 2015 stereo benchmark [39], as well as the Middlebury 2014 stereo benchmark [40]. In this, we have also evaluated and studied the effects of different configurations of the processing pipeline for real-time disparity estimation, e.g., the effect of reducing the number of SGM paths or the improvement of the additional subpixel refinement. We compare the results of our implementations with state-of-the-art approaches and analyze their performance with respect to the power consumption of the embedded system.

In the second part (Section 3.2), we qualitatively discuss the results of the use-case specific experiments we have conducted. In this, we have deployed our approaches on a low-cost UAV and performed real-time disparity estimation based on a stereo camera system, which is pointing forward in the direction of flight. The resulting disparity map can then be used for reactive obstacle avoidance as proposed in our previous work [19] or for facade and close-range object reconstruction.

3.1. Quantitative Evaluation of Accuracy on Public Stereo Benchmarks

The quantitative assessment of the performance of our approaches comprises an evaluation with respect to their accuracy in Section 3.1.1, as well as studies on the effects of the subpixel disparity refinement in Section 3.1.2 and the improvements gained by an accurate left–right consistency check in Section 3.1.3. For the evaluation of the accuracy of our approach and its different configurations, we have used the training set of the KITTI 2015 stereo benchmark [39], which consists of 200 stereo image pairs and ground truth disparity maps captured by a LiDAR sensor from on top of a car driving around urban areas, as well as the Middlebury 2014 stereo benchmark [40]. The latter one allows a more thorough evaluation on the accuracy of our approach, and the effects of different optimizations in the processing pipeline, since it consists of 15 high-resolution stereo pairs of indoor scenes, together with highly accurate and dense ground truth disparity maps captured by a structured light sensor. Moreover, in Section 3.1.4, we study the processing speed and power consumption of our approaches, together with the effects of reducing the aggregation paths of the SGM optimization. For our experiments, we have deployed our approaches on the NVIDIA Jetson Xavier AGX with an 8-core 64 bit ARMv8.2 CPU and a 512-core Volta GPU. All measurements with respect to accuracy, timings and power consumption were done on this hardware.

The standard evaluation routine of the KITTI 2015 stereo benchmark [39] states the accuracy as the number of erroneous pixels (D1-all), averaged over all m ground truth pixels in the evaluation set, for which the estimated disparity d differs by 3 or more pixels with respect to the ground truth $\hat{d}$:

$$\text{D1-all} = \frac{1}{m} \sum_{i=1}^{m} \left[|d - \hat{d}| \geq 3 \right], \tag{6}$$

with $[\cdot]$ being the Iverson bracket. Since the ground truth was generated from a LiDAR sensor, mounted at a slightly different position as the camera, for which the disparity map is estimated, the ground truth also provides disparity values in areas which are occluded in the second camera image and, in turn, usually only contains limited information in the estimated disparity map. Although the KITTI benchmark also provides ground truth maps which only contain non-occluded (noc) areas, the standard evaluation protocol uses the occluded (occ) dataset, which we have also used for our evaluation in Table 2. Furthermore, the benchmark distinguished between the results of the actual estimated (Est) disparity maps and interpolated versions of them (All). The latter ones allow a comparison between disparity maps of different density, by applying a background interpolation to fill the pixels in the estimated disparity map for which no data are available. However, since our approach uses a left–right consistency check and a median filter to explicitly remove outliers and inconsistent areas, we are more interested in the results achieved by the actual estimate. Nonetheless, for comparison, we also provide the results achieved by the interpolated disparity maps, as well as the information on the density of the non-interpolated map, if available, which states the number of pixels in the estimated disparity map which contain valid estimates.

Similar to the evaluation routine of the KITTI 2015 stereo benchmark, the Middlebury benchmark ranks the algorithms based on four different accuracy levels, namely the amount of pixels whose error is greater than 0.5 (bad0.5), 1 (bad1), 2 (bad2) and 4 (bad4) pixels with respect to all m ground truth pixels in the evaluation set:

$$\text{bad}\theta = \frac{1}{m} \sum_{i=1}^{m} \left[|d - \hat{d}| > \theta \right], \tag{7}$$

with d and $\hat{d}$ again denoting the estimated and ground truth disparity respectively, and $[\cdot]$ being the Iverson bracket. The data are provided in full (F) image resolution with up to 3000×2000 pixels and a disparity range of up to 800 pixels, as well as half (H) and quarter (Q) image resolution. The official evaluation is always performed on the full image resolution. Thus, if the results are generated on a dataset with a smaller resolution, the results are first being up-sampled before being evaluated.

3.1.1. Accuracy

In this section, we will first list the accuracy achieved on the KITTI 2015 stereo benchmark, which is followed by the evaluation of the accuracy on the Middelbury 2014 stereo benchmark.

Table 2. Accuracy achieved by different algorithms and configurations on the KITTI 2015 stereo benchmark [39]. Although the upper part lists algorithms that are optimized and deployed on embedded hardware, the three algorithms at the bottom are listed as a reference and a baseline. The results achieved by different configurations of our implementation are listed in the middle section. The accuracy is stated as the number of erroneous pixels (D1-all), for which the estimated disparity differs by 3 or more pixels with respect to the ground truth. The KITTI 2015 benchmark distinguishes between the result of the actual estimated (Est) disparity map and an interpolated version of it (All), in which the pixels, for which no disparity is available, become interpolated by a simple background interpolation. As an evaluation ground truth, we have considered all available pixels, not only the non-occluded ones. The density indicates, how many pixels inside the computed disparity maps have an estimate. †: The accuracy stated is computed with respect to the non-occluded pixels in the ground truth.

Approach	Configuration	HW Device	Resolution (in pixels)	Accuracy		
				D1-all (Est.)	D1-all (All)	Density
Zhao et al. [18]	$CT_{5\times5}$—SGM	FPGA	1242 × 375	-	11.8%	-
Zhao et al. [18]	$CT_{7\times7}$—SGM	FPGA	1242 × 375	-	9.5 %	-
Ruf et al. [19]	$CT_{5\times5}$—SGM	FPGA	640 × 360	4.6%	31.2%	46.4%
Rahnama et al. [17]	$CT_{5\times5}$—MGM	FPGA	1242 × 375	6.7%	13.6%	81.0%
Rahnama et al. [17]	$CT_{13\times13}$—MGM	FPGA	1242 × 375	4.8%	9.9%	85.0%
Cui and Dahnoun [41] †	$NCC_{9\times9}$-native	GPU	1242 × 375	-	16.6%	-
Cui and Dahnoun [41] †	$NCC_{9\times9}$-optimized	GPU	1242 × 375	-	13.1%	-
Chang et al. [26]	Z^2-ZNCC	GPU	1242 × 375	7.6%	7.7%	99.9%
Hernandez-Juarez et al. [25]	$CT_{9\times7}$—SGM	GPU	1242 × 375	8.2%	8.2%	100%
ReS^{2}tAC—CUDA	$CT_{5\times5}$—SGM	GPU	640 × 480	5.4%	8.4%	94.5%
ReS^{2}tAC—CUDA	$CT_{5\times5}$—SGM	GPU	1242 × 375	4.3%	8.3%	88.8%
ReS^{2}tAC—CUDA	$CT_{9\times7}$—SGM	GPU	640 × 480	5.1%	7.9%	94.6%
ReS^{2}tAC—CUDA	$CT_{9\times7}$—SGM	GPU	1242 × 375	4.0%	7.7%	90.0%
ReS^{2}tAC—CUDA	$NCC_{5\times5}$—SGM	GPU	640 × 480	5.3%	7.8%	94.8%
ReS^{2}tAC—CUDA	$NCC_{5\times5}$—SGM	GPU	1242 × 375	4.3%	8.1 %	90.0%
ReS^{2}tAC—CUDA	$NCC_{9\times9}$—SGM	GPU	640 × 480	5.9%	8.2 %	94.7%
ReS^{2}tAC—CUDA	$NCC_{9\times9}$—SGM	GPU	1242 × 375	4.8%	8.3 %	91.1 %
ReS^{2}tAC—NEON	$CT_{5\times5}$—SGM	CPU	640 × 480	5.0%	7.9%	94.5%
ReS^{2}tAC—NEON	$CT_{5\times5}$—SGM	CPU	1242 × 375	4.6%	8.5%	90.0%
Schönberger et al. [42]	$NCC_{7\times7}$—SGM-Forest	CPU	1242 × 375	4.3%	4.4%	99.9%
OpenCV-SGBM	$SAD_{3\times3}$—SGM	CPU	1242 × 375	5.9%	10.9%	90.4%
Hirschmueller [8]	CT—SGM	GPU	1242 × 375	6.4%	6.4%	100%

KITTI 2015 Stereo Benchmark

Table 2 lists the quantitative results of the accuracy of different configurations of our implementations, as well as those of other approaches and implementations, which are achieved on the KITTI 2015 stereo benchmark. Although the results of our approach were achieved on the training set of the benchmark, the results of the approaches from literature were taken either from the corresponding publication or the official listing of the benchmark, which lists the results achieved on the actual test set. The upper part of Table 2 lists algorithms and configurations, which are optimized for the deployment and execution on embedded hardware. Not all of these are variants of the Semi-Global Matching stereo algorithm. Yet, they serve as a good comparison since they were deployed on the same or similar hardware as ours. The three algorithms at the bottom of the list serve as a baseline to our approaches. Although the one from Hirschmueller [8] reveals the accuracy achieved by the original SGM algorithm implemented on the GPU with the census transform of unknown size as a cost function, the SGBM variant of the OpenCV library is widely spread and easy to use, but not optimized for embedded processing. The algorithm of Schönberger et al. [42] is evaluated on both the KITTI 2015 stereo benchmark, as well as the Middlebury 2014 stereo benchmark. They propose to use a random forest classifier to learn to efficiently fuse the different scanline optimizations of the SGM algorithm, to reduce the number of optimization paths for embedded processing more efficiently.

We have measured the accuracy of our approach using different cost functions and different support regions. For each configuration, we have evaluated the results achieved on the original image size provided by the KITTI benchmark, i.e., 1242 × 375 pixels, as well as on images with VGA resolution. In the case of VGA resolution, we have down-sampled the original images to a resolution of 640 × 480 pixels, performed the stereo disparity estimation and up-sampled the resulting disparity maps to the original image size with a nearest-neighbor interpolation and a scaling of the disparities by the horizontal scale factor. Thus, we have always used the original image size for the accuracy measurements. For the selection of the SGM penalties, we have empirically evaluated different values and selected those with the best result. In terms of the best configurations, these are $P_1 = 27$ and $P_2 = 86$ for the $CT_{9\times7}$ and $P_1 = 90$ and $P_2 = 880$ for the $NCC_{5\times5}$. An excerpt on the qualitative results for the best configuration of our approach is presented in Figure 11.

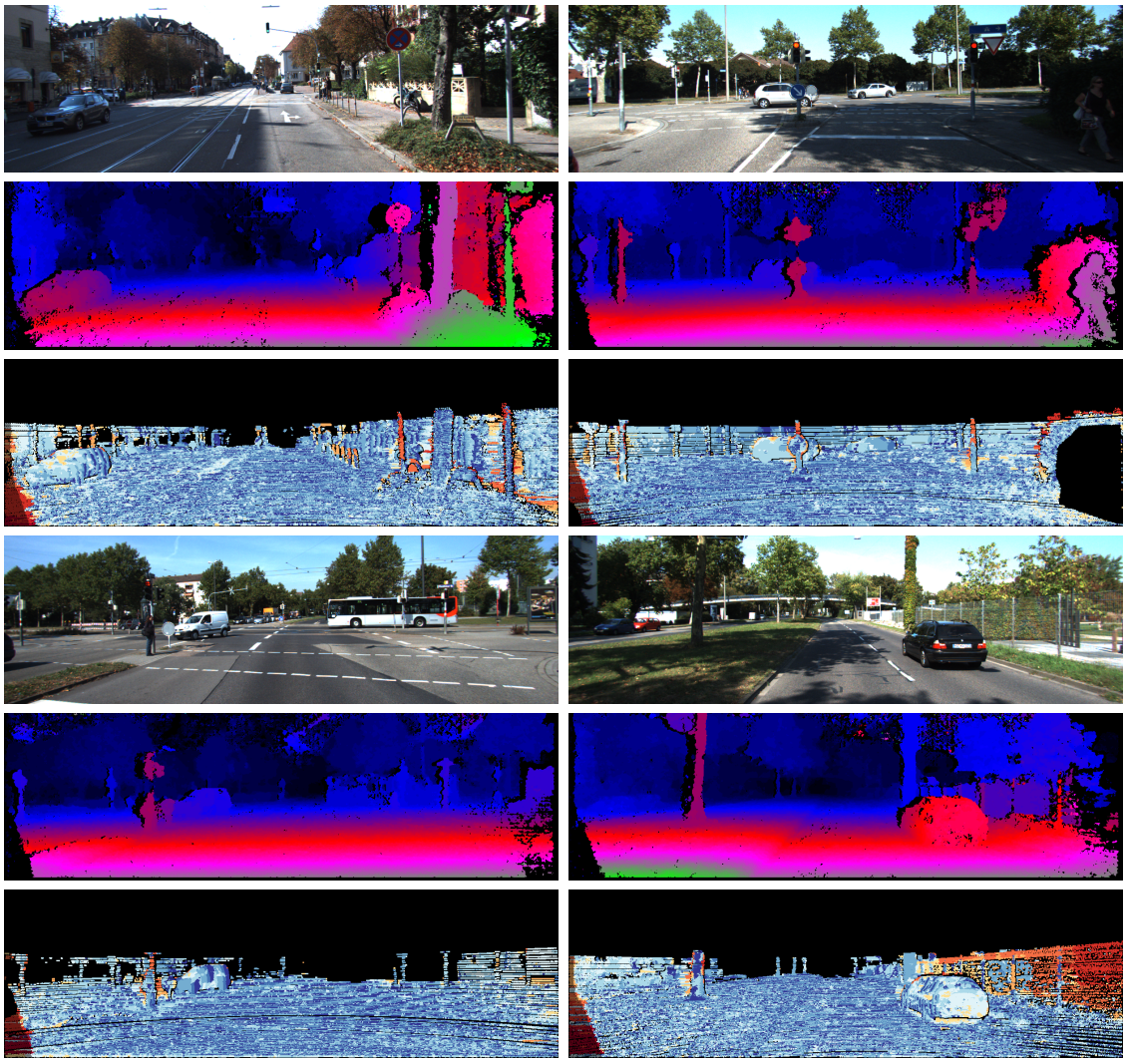


Figure 11. Four exemplary results from the KITTI 2015 stereo benchmark, computed with the $CT_{9\times7}$—SGM configuration on the original image size. **Rows 1 & 4**: Reference image. **Rows 2 & 5**: Estimated disparity maps. **Rows 3 & 6**: Color-coded error image between the prediction and the ground truth. Error images use a log-color scale as described by Menze and Geiger [39], marking correct pixels in estimates and wrong estimates in red color tones.

The results in Table 2 reveal that the use of the census transform with a support region of 9 × 7 pixels and its Hamming distance as a matching cost function, achieves the best results, in the evaluation of both the actual estimate and the interpolated version. As can be expected, the use of down-sampled versions of the input images yields less accurate results

and yet, achieves a higher density in the resulting disparity maps. Furthermore, the use of the normalized cross-correlation as a cost function achieves slightly less accurate results, while leading to a smaller throughput as discussed in Section 3.1.4. The implementation on the CPU using NEON SIMD intrinsics with a CT of size 5×5 pixels achieves similar and, in the case of the smaller image resolution, slightly better results than its GPU counterpart. In summary, when considering the actual estimate, our configuration $CT_{9\times7}$—SGM executed on the GPU, with the original image size of 1242×375 pixels, outperforms the baseline implementations as well as the other algorithms optimized for embedded hardware. When evaluating on the interpolated disparity maps, our approaches achieve similar and mostly better results than the other embedded algorithms. The superiority of the baseline algorithms from Hirschmueller [8] and Schönberger et al. [42] with respect to the quality of the disparity estimation is to be expected, since they were optimized with respect to accuracy and not speed or throughput.

Middelbury 2014 Stereo Benchmark

The results achieved by our approach on the training set of the benchmark are listed in Table 3, with a qualitative excerpt of the results achieved by the best configuration presented in Figure 12. None of the other approaches for stereo processing on embedded hardware, which were listed in the evaluation on the KITTI benchmark, have also been evaluated on the Middlebury 2014 stereo benchmark and, thus, these are not listed in this evaluation. However, results on the non-embedded baseline algorithms are available and are again listed in the lower part of Table 3. We have evaluated the same configurations as in the evaluation on the KITTI benchmark. Since our approach can only handle a disparity range of up to 256 pixels, we computed the disparity maps on the provided quarter image resolution (Orig. Q). Just as in the standard evaluation routine of the benchmark, the results listed were found after up-sampling the disparity maps to the original image resolution with a nearest-neighbor interpolation and scaling the containing disparities with a factor of 4. Again, for the selection of the SGM penalties, we have empirically evaluated different values and selected those which yield the best results, being $P_1 = 11$ and $P_2 = 39$ for the $CT_{5\times5}$ and $P_1 = 140$ and $P_2 = 730$ for the $NCC_{5\times5}$.

Table 3. Accuracy achieved by different algorithms and configurations on the Middlebury 2014 stereo benchmark [40]. Although the upper part lists the results of different configurations of our implementation, the three algorithms at the bottom are listed as a reference and a baseline. The accuracy is stated as the number of erroneous pixels, whose error is greater than 0.5 (bad0.5), 1 (bad1), 2 (bad2) and 4 (bad4) pixels with respect to the ground truth. The density indicates, how many pixels inside the computed disparity maps have an estimate. The Middlebury 2014 stereo benchmark provides the image data in full (F), half (H) and quarter (Q) image resolution. We have computed our results on the quarter image resolution and evaluated them according to the standard evaluation pipeline on the full resolution.

Approach	**Configuration**	**Resolution**	**Accuracy**				
			bad0.5	**bad1**	**bad2**	**bad4**	**Density**
ReS^{2}tAC—CUDA	$CT_{5\times5}$—SGM	Orig. Q	77.0%	59.5%	35.4%	13.5%	93.0%
ReS^{2}tAC—CUDA	$CT_{9\times7}$—SGM	Orig. Q	77.3%	59.9%	35.7%	13.6%	93.3%
ReS^{2}tAC—CUDA	$NCC_{5\times5}$—SGM	Orig. Q	77.1%	60.1%	36.3%	14.4%	92.8%
ReS^{2}tAC—CUDA	$NCC_{9\times9}$—SGM	Orig. Q	77.1%	61.2%	38.7%	17.1%	91.9%
ReS^{2}tAC—NEON	$CT_{5\times5}$—SGM	Orig. Q	76.2%	59.0%	35.1%	13.4%	92.1%
Schönberger et al. [42]	$NCC_{7\times7}$—SGM-Forest	Orig. H	43.1%	14.8%	7.0%	3.7%	100%
OpenCV-SGBM	$SAD_{3\times3}$—SGM	Orig. Q	67.3%	42.1%	25.5%	17.3%	100%
Hirschmueller [8]	CT—SGM	Orig. H	51.5%	28.2%	17.7%	12.2%	100%

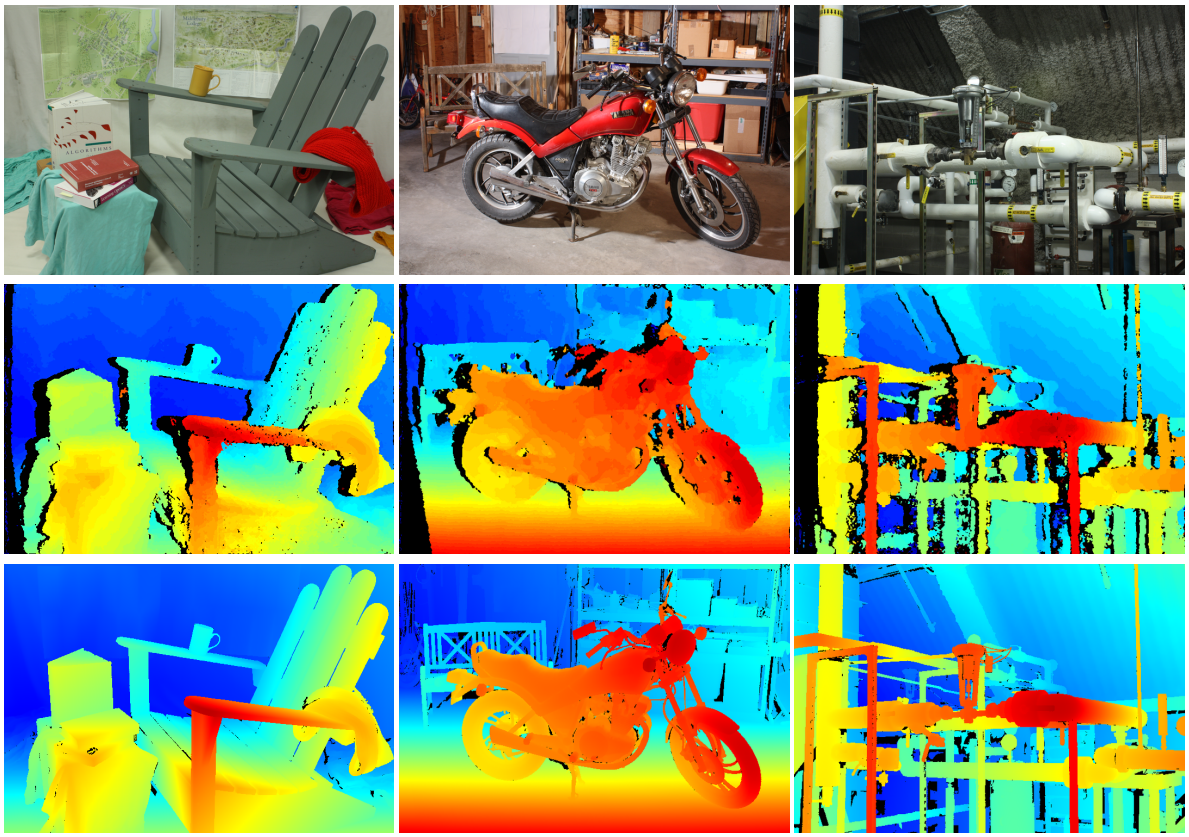


Figure 12. Three exemplary results from the Middlebury 2014 stereo benchmark, computed with the best performing base configuration, i.e., $CT_{5\times5}$-SGM on the quarter image resolution. **Row 1**: Reference image. **Row 2**: Estimated disparity map. **Row 3**: Ground truth disparity map.

Table 3 does not yield any satisfying results. This, however, is not surprising, since we have only used quarter of the image resolution to compute the results and up-sampled them by a factor of 4 for evaluation, introducing a lot of errors due to interpolation. As stated by Scharstein et al. [40], the aim of this benchmark is on providing new challenges for modern stereo algorithms in terms of image resolution, accuracy and scene complexity, and not necessarily on the evaluation of optimizations with respect to computational efficiency and run-time. Nonetheless, the accuracy in the ground truth and the evaluation protocol of the Middlebury 2014 stereo benchmark allows for an evaluation of the improvement gained by a subpixel disparity refinement as done in the following section.

3.1.2. The Effect of Subpixel Disparity Refinement

As described in Section 2.1.3, a subpixel disparity refinement can be computed for each pixel in the disparity map, by fitting a parabola through the matching costs of the winning disparity and its two neighbors. This achieves an increase in accuracy of up to 0.8% in the case of the KITTI benchmark (Table 4), and up to 9% in the case of the Middlebury benchmark (Table 5), and yet only requires a small computational overhead (cf. Table 6).

Table 4. Accuracy achieved by selected configurations of our approach with an additional subpixel disparity refinement on the training set of the KITTI 2015 stereo benchmark [39]. The corresponding differences to the accuracies listed in Table 2 are given in parentheses.

Approach	Configuration	Resolution (in pixels)	Accuracy	
			D1-all (Est.)	D1-all (All)
ReS^{2}tAC—CUDA	$CT_{5\times5}$—SGM—fine	1242 × 375	3.5% (−0.8)	8.0% (−0.4)
ReS^{2}tAC—CUDA	$CT_{9\times7}$—SGM—fine	1242 × 375	3.3% (−0.7)	7.4% (−0.3)
ReS^{2}tAC—CUDA	$NCC_{5\times5}$—SGM—fine	1242 × 375	3.7% (−0.6)	7.7% (−0.4)
ReS^{2}tAC—CUDA	$NCC_{9\times9}$—SGM—fine	1242 × 375	4.3% (−0.5)	7.9% (−0.4)

Table 5. Accuracy achieved by selected configurations of our approach with an additional subpixel disparity refinement on the training set of the Middlebury 2014 stereo benchmark [40]. The corresponding differences to the accuracies listed in Table 3 are given in parentheses.

Approach	Configuration	Resolution	Accuracy			
			bad 0.5	bad 1	bad 2	bad 4
ReS^{2}tAC—CUDA	$CT_{5\times5}$—SGM—fine	Orig. Q	72.4% (−4.6)	52.1% (−7.4)	26.1% (−9.3)	10.4% (−3.1)
ReS^{2}tAC—CUDA	$CT_{9\times7}$—SGM—fine	Orig. Q	73.0% (−4.3)	52.7% (−7.2)	26.6% (−9.1)	10.7% (−2.9)
ReS^{2}tAC—CUDA	$NCC_{5\times5}$—SGM—fine	Orig. Q	73.8% (−3.3)	53.8% (−6.3)	27.8% (−8.5)	12.1% (−2.0)
ReS^{2}tAC—CUDA	$NCC_{9\times9}$—SGM—fine	Orig. Q	73.8% (−3.3)	55.2% (−6.0)	30.6% (−8.1)	14.8% (−2.3)

3.1.3. Accurate Left–Right Consistency Check

To evaluate the effects of only approximating the disparity map corresponding to the right image of the stereo pair, which is needed for the left–right consistency check (cf. Section 2.1.3), we have implemented a more exact and more costly consistency check for the GPU. In this, we switch and flip the reference and matching image and calculate a second disparity map for the original matching image. This leads to a more accurate disparity map D^R for the right input image, which, in turn, is used by Equation (A1) of the consistency check. With a more accurate D^R it is assumed that the consistency check is more effective in filtering outliers, but does the high computational overhead of fully calculating two disparity maps justify the increase in accuracy? The results of our evaluation do not indicate a significant improvement. Our studies reveal an increase in accuracy of only 0.4–1.0%, when calculating the disparity map of the right image from scratch compared to only approximating it from the cost volume corresponding to the left disparity map. However, the throughput is nearly halved when using a more accurate consistency check, as illustrated by configurations with the prefix *exact-cc* (exact consistency check) in Table 6.

3.1.4. Throughput, Frame Rates and Power Consumption

A typical measure to quantify the processing speed of a stereo algorithm is the number of frames per second (FPS) which can be calculated. However, since the FPS greatly depends on the image size of the output and the disparity range, we instead reason on the efficiency of our approaches based on their throughput, which is measured in million disparity estimations per second (MDE/s):

$$\text{MDE/s} = \frac{W \cdot H \cdot |\Gamma|}{\text{run-time}}, \tag{8}$$

with W and H being the width and the height of the resulting disparity map, and $|\Gamma|$ being the size of the disparity range. Given the throughput achieved by a certain configuration or algorithm, it is possible to deduce the expected frame rates for a set of image size and disparity range:

$$\text{FPS} = \frac{\text{MDE/s}}{W \cdot H \cdot |\Gamma|}, \tag{9}$$

as done in Figure 13. Furthermore, the throughput allows for a better comparison between different algorithms with respect to their processing speed, due to its independence of a fixed image size and disparity range.

In Table 6, we have listed the throughput achieved by different configurations of our approach, as well as the throughput of selected embedded algorithms from literature. Although all measurements of our approach were done on the NVIDIA Jetson Xavier AGX with maximum performance, the related work optimized for the execution on GPU were deployed on the NVIDIA Jetson TX1 and TX2. In the case of the related work, which has not explicitly stated the throughput of the respective algorithm, we have used the frame rate and the corresponding image size to calculate the values according to Equation (9). For the measurement of the run-time of our approaches, we have considered the whole pipeline, including the upload of the stereo image pair and the download of the disparity image to and from the device memory of the GPU.

First, the measurements reveal the higher computational efficiency of the census transform as a matching cost function with respect to the normalized cross-correlation. And secondly, they show the small computational overhead of the subpixel disparity refinement (fine), as discussed in Section 3.1.2. However, the measurements also unveil that our optimizations are less efficient than those that can be found in the literature. As expected, the implementations which are optimized and deployed on FPGA architectures are superior to those running on an embedded GPU. Furthermore, the superiority in terms of throughput of algorithms, such as those from Cui and Dahnoun [41] and Chang et al. [26] that do not rely on a complex regularization scheme such as the SGM is also to be expected. Nonetheless, our approach has a lower throughput than a similar implementation of Hernandez-Juarez et al. [25], while simultaneously being deployed on a more powerful system.

Table 6. Throughput achieved by our approach and selected embedded algorithms from literature. The throughput is measured in million disparity estimations per second (MDE/s). All the measurements for our approach were done on the NVIDIA Jetson Xavier AGX board, with the power setting set to maximum performance.

Approach	Configuration	HW Device	Throughput (in MDE/s)
Zhao et al. [18]	$CT_{5\times5}$—SGM	FPGA	9589.9
Zhao et al. [18]	$CT_{7\times7}$—SGM	FPGA	8743.7
Ruf et al. [19]	$CT_{5\times5}$—SGM	FPGA	400.8
Rahnama et al. [17]	$CT_{5\times5}$— MGM	FPGA	4246.9
Cui and Dahnoun [41]	$NCC_{9\times9}$-optimized	GPU (TX2)	6497.1
Chang et al. [26]	Z^2-ZNCC	GPU (TX2)	1669.2
Hernandez-Juarez et al. [25]	$CT_{9\times7}$—SGM	GPU (TX1)	747.1
ReS^{2}tAC-CUDA	$CT_{5\times5}$—SGM	GPU (AGX)	652.7
ReS^{2}tAC-CUDA	$CT_{9\times7}$—SGM	GPU (AGX)	644.9
ReS^{2}tAC-CUDA	$CT_{5\times5}$—SGM—fine	GPU (AGX)	640.9
ReS^{2}tAC-CUDA	$CT_{9\times7}$—SGM—fine	GPU (AGX)	633.1
ReS^{2}tAC-CUDA	$CT_{5\times5}$—SGM—exact-cc	GPU (AGX)	365.7
ReS^{2}tAC-CUDA	$CT_{9\times7}$—SGM—exact-cc	GPU (AGX)	361.8
ReS^{2}tAC-CUDA	$NCC_{5\times5}$—SGM	GPU (AGX)	442.4
ReS^{2}tAC-CUDA	$NCC_{9\times9}$—SGM	GPU (AGX)	344.1
ReS^{2}tAC-NEON	$CT_{5\times5}$—SGM	CPU	166.2

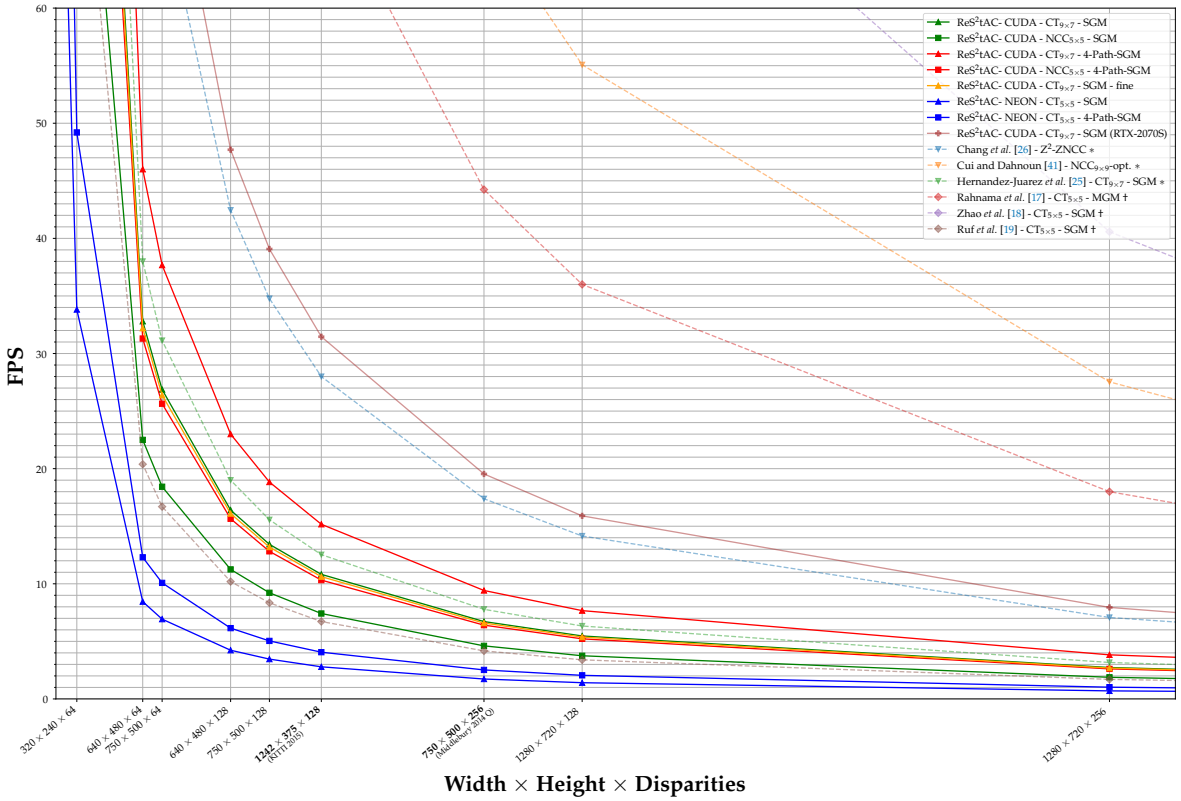


Figure 13. Expected frames per second (FPS) for sets of different image sizes and disparity ranges, based on the throughput achieved by different configurations and approaches listed in Tables 6 and 7. Image resolution and disparity ranges corresponding to the KITTI 2015 and Middlebury 2014 Q benchmark are printed in bold. ∗: Approaches from literature deployed on embedded GPU hardware. †: Approaches from literature deployed on embedded FPGA hardware.

A common way to further increase the throughput of the SGM algorithm is to reduce the number of aggregation paths. Most of the implementations aggregated the matching costs for each pixel along eight concentric paths. However, studies [10,25] suggest that a reduction of the aggregation paths from eight to four does not have a significant negative impact on the accuracy of the resulting disparity map, while greatly increasing the processing speed. This is also supported by our experiments, in which we have left out the diagonal aggregation paths of the SGM optimization, since they are the longest ones, and only regularized the cost volume with the two horizontal and the two vertical aggregation paths. The results of our experiments are listed in Table 7, showing an increase in the throughput by a factor of up 1.45, while reducing the accuracy and the disparity by a maximum of 0.2% and 1.6% respectively. In this, we outperform the approach of Hernandez-Juarez et al. [25], which is comparable to ours, in terms of accuracy and throughput.

With the throughput listed in Tables 6 and 7, we calculated the expected FPS, which are to be achieved for sets of different image sizes and disparity ranges according to Equation (9), and plotted these as curves in Figure 13. In this, we have not plotted all our configurations, but selected one for each cost function and hardware, as well as the corresponding versions with only four paths in the SGM optimization. Additionally, we have selected one configuration that performs a subpixel disparity refinement for comparison. We have also plotted the FPS curves for the related approaches from literature deployed on FPGA (†) and GPU (∗) architectures, as well as one curve achieved by our approach on a high-end desktop NVIDIA RTX 2070 Super GPU. We have printed the image sizes and disparity ranges corresponding to the KITTI 2015 and Middlebury 2014 Q benchmark in bold. The curves provide a good visual representation of the throughput listed in Tables 6 and 7 and show that with decreasing complexity, i.e., a reduced image size

and disparity range, the frame rates increase rapidly. Thus, the curves reveal how different configurations, approaches and used hardware compare in terms of processing speed.

Table 7. Throughput and accuracy achieved with a reduced number of aggregation paths in the SGM optimization. Instead of eight aggregation paths, only the two horizontal and the two vertical paths were used. The accuracy in terms of error rate and density was measured on the KITTI 2015 stereo benchmark.

Approach	Configuration	HW Device	Throughput (in MDE/s)	Accuracy	
				D1-all (Est.)	Density
ReS^{2}tAC-CUDA	$CT_{5\times5}$—4-Path-SGM	GPU (AGX)	924.1 (×1.42)	4.3% (+0.0)	88.1% (−0.7)
ReS^{2}tAC-CUDA	$CT_{9\times7}$—4-Path-SGM	GPU (AGX)	904.4 (×1.40)	4.2% (+0.2)	89.2% (−0.8)
ReS^{2}tAC-CUDA	$NCC_{5\times5}$—4-Path-SGM	GPU (AGX)	615.4 (×1.39)	4.3% (+0.0)	88.6% (−1.4)
ReS^{2}tAC-CUDA	$NCC_{9\times9}$—4-Path-SGM	GPU (AGX)	436.5 (×1.27)	4.6% (+0.2)	89.5% (−1.6)
ReS^{2}tAC-NEON	$CT_{5\times5}$—4-Path-SGM	CPU	241.8 (×1.45)	4.8% (+0.2)	89.3% (−0.7)

A key characteristic of embedded systems is their power consumption and, depending on what platform the system is deployed, this can be a very crucial characteristic. The simple metric of FPS per watt (FPS/W) helps to quantify the efficiency of image processing algorithms with respect to the power consumption of the system on which they are deployed. The NVIDIA Jetson Xavier AGX, on which we have deployed our approach, allows the setting of four different power settings, namely:

MAXN This is the setting enabling the maximum performance. With this, all eight cores of the ARM CPU are activated and can clock up to a maximum of 2.3 GHz. The maximum clock rate of the GPU is set to 1.4 GHz. This is the setting with which all previous experiments were conducted.

30 W In this, again all eight cores of the CPU are enabled. However, they are restricted to a maximum clock rate of 1.2 GHz. Furthermore, the clock rate of the GPU is restricted to 905 MHz.

15 W In this setting, four cores of the CPU are enabled which clock at a maximum rate of 1.2 GHz, while the GPU clocks up to 675 MHz.

10 W In the smallest setting, only two cores of the CPU are enabled with a maximum of 1.2 GHz and the clock rate of the GPU is restricted to only 522 MHz.

In Figure 14, we have plotted the FPS/W which are expected to be achieved by different configurations of our approach, as well as by some approaches form the literature, again, depending on different image sizes and disparity ranges. These calculations are based on the throughput and power consumption measured or stated. In terms of our approach, we have selected the two configurations reaching the highest throughput on the GPU and the CPU. We have varied the power setting and measured the throughput and power consumption, the latter being provided by internal sensors of the AGX. Please note that the actual power consumption on the AGX does not coincide with the statement of the power setting, as the latter one only indicates an upper bound on the consumption. From literature, we have selected approaches which provide a value on the power consumption in addition to the throughput or frame rate. Unfortunately, in terms of related work that has deployed stereo algorithms on embedded GPUs, this was only done by Hernandez-Juarez et al. [25].

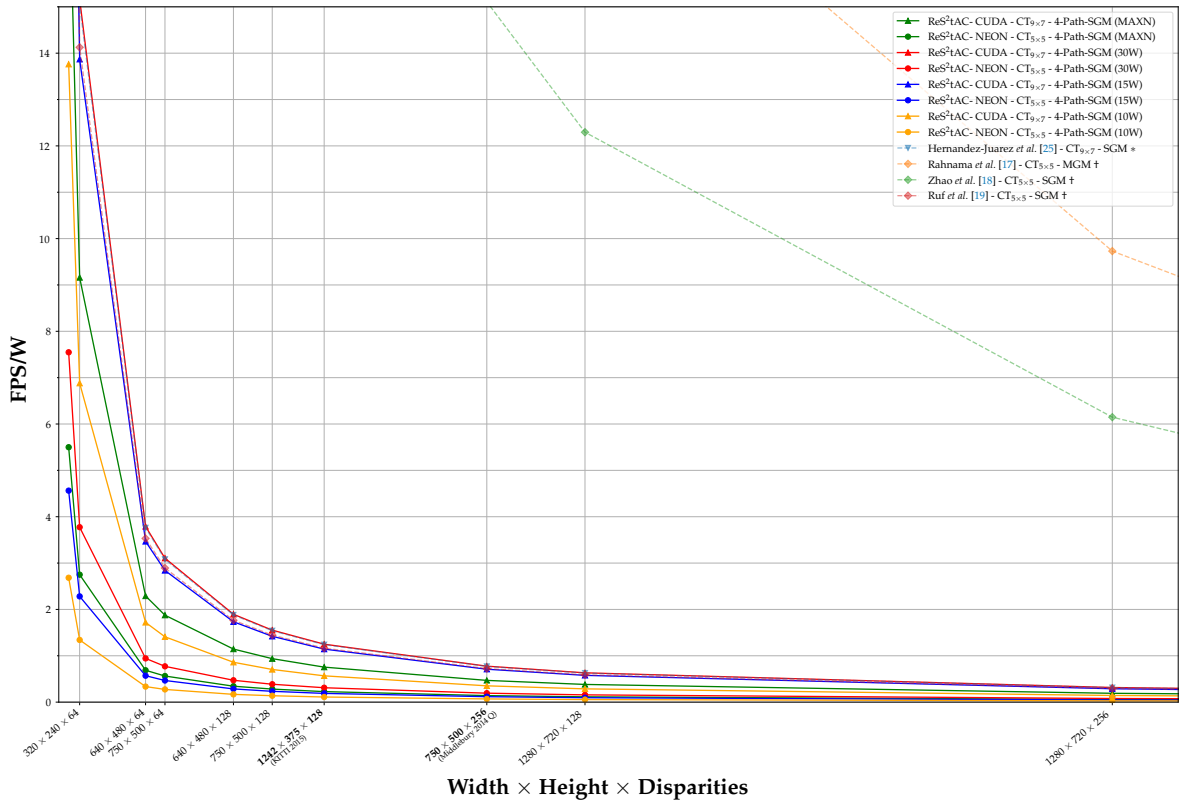


Figure 14. Expected FPS/W for sets of different image sizes and disparity ranges, based on the throughput and the power consumption achieved by different configurations and approaches. In terms of our approach, the power settings of the AGX were varied to measure its efficiency. Image resolution and disparity ranges corresponding to the KITTI 2015 and Middlebury 2014 Q benchmark are printed in bold. ∗: Approaches from literature deployed on embedded GPU hardware. †: Approaches from literature deployed on embedded FPGA hardware.

The curves in Figures 13 and 14 clearly reveal the superiority of FPGA-based approaches over those deployed on GPUs. Not only do they achieve much higher frame rates, but also require significantly less power and, in turn, also achieve higher FPS/W. However, the emerging embedded GPUs also achieve quite reasonable frame rates with respect to their power consumption, and depending where the systems are deployed, e.g., quadrotor-based systems, the power required by the GPU is negligible, when compared to that required by the rotors. However, more on this is provided in the discussion on what the results mean for our use-case (cf. Section 4). Interestingly, the curves in Figure 14 show that both the CUDA and NEON implementation of our approach achieve the best efficiency on the 30 W power setting. Even the CUDA implementation being run on the 15 W power setting is still more efficient than the same run on the setting with maximum performance. We assume that this is the result of clocking down the CPU, since it is less efficient than the GPU. Nonetheless, the 30 W and 15 W power setting reduce the throughput by approximately 29% and 42%, respectively, compared to that achieved on MAXN.

3.2. Qualitative and Quantitative Evaluation of Real-Time Stereo Processing on Board Low-Cost UAVs

As part of our use-case specific experiments, in which we want to bring real-time stereo processing with an embedded CUDA device on board a low-cost UAV, we have equipped a DJI Matrix 210v2 RTK with a DJI Manifold 2-G processing unit, which is based on the NVIDIA Jetson TX2 architecture equipped with a 4-core 64 bit ARMv8 CPU and a 256-core Pascal GPU. As a stereo camera we have used the integrated stereo vision sensor, which can be accessed by the Manifold through the DJI onboard SDK. Our setup is illustrated in Figure 15.

Figure 15. The system used for use-case specific experiments for real-time stereo processing with an embedded CUDA device on board a low-cost UAV: A DJI Matrix 210 v2 RTK equipped with a DJI Manifold 2-G processing unit. The integrated stereo vision sensor is used as stereo camera.

In the maximum power setting (MAXN), the ARM Cortex A57 CPU of the TX2 inside the Manifold has four cores with a maximum clock rate of 2 GHz, while the built-in Tegra GPU clocks up to a maximum of 1.3 GHz. The integrated vision sensor provides non-rectified, grayscale stereo image pairs with an image resolution of up to 640 × 480 pixels at a frame rate of 20 FPS. We have used the standard calibration routine of OpenCV to calibrate the stereo sensor and, in turn, precompute the rectification maps, needed to transform the input images into a rectified stereo pair prior to the actual stereo processing. The integrated stereo vision sensor also provides precomputed disparity maps at a frame rate of 10 FPS and with an image resolution of 320 × 240 pixels [43].

We have deployed and tested two configurations of our approach, one running on the GPU and the other on the CPU, namely: ReS^{2}tAC-CUDA with $CT_{9\times7}$—4-Path-SGM and ReS^{2}tAC-NEON with $CT_{5\times5}$. Both configurations rely on the Hamming distance of the census transform as a cost function and use only four paths in the SGM optimization to reach a higher throughput. The throughput and frame rates, as well as qualitative results achieved by the two configurations are listed in Table 8 and Figure 16 respectively.

Table 8. Throughput and frame rate achieved by two configurations on the DJI Manifold, equipped with a Jetson TX2.

Approach	Configuration	HW Device	Throughput (in MDE/s)	Frame Rate	
				at 640 × 480 × 64 pixels (in FPS)	at 320 × 240 × 64 pixels (in FPS)
ReS^{2}tAC-CUDA	$CT_{9\times7}$—4-Path-SGM	GPU (TX2)	304.7	15.5	62.0
ReS^{2}tAC-NEON	$CT_{5\times5}$—4-Path-SGM	CPU	102.2	5.2	20.8

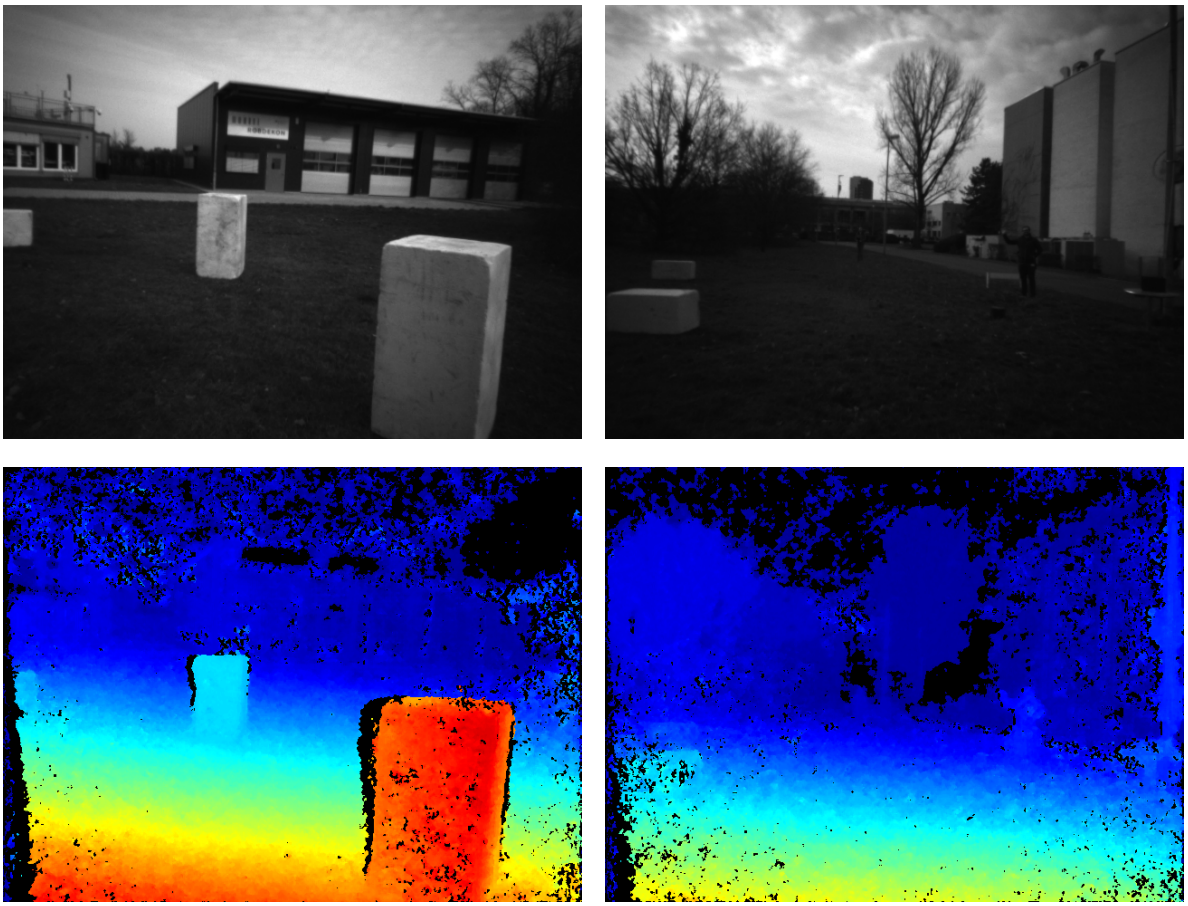


Figure 16. Qualitative results of our approach run on the DJI Manifold 2-G on top of the DJI Matrix 210v2 RTK, using the data of the stereo vision sensor as input. In this, the UAV was flying 1–2 m above the ground to demonstrate the ability of our approach to appropriately estimate the scene depth. **Top row:** Rectified reference images. **Bottom row:** Corresponding disparity maps, color-coded with the jet color map, going from red (near), over yellow and green to blue (far).

4. Discussion

In the following, we will discuss the findings of our experiments with respect to three different aspects, namely accuracy (Section 4.1), processing speed (Section 4.2) and power consumption (Section 4.3) in the light of possible applications, before finishing this paper with a short summary, a conclusion and an outlook.

4.1. Accuracy

The quantitative evaluation on the KITTI 2015 stereo benchmark (Table 2) reveals a high and state-of-the-art accuracy of our approach, both in the actual estimated disparity map (Est), in which inconsistent regions are removed, as well as the interpolated versions (All). The latter are being used as part of the standard evaluation routine of the benchmark. Understandably, the accuracy of the interpolated disparity maps is used as a primary ranking in the benchmark, since it allows the comparison of disparity maps with different densities; however, in the assessment of the performance of our approach, the accuracies of the filtered disparity maps are of greater importance. In particular, when looking at the qualitative results (Figure 11) and the stated densities of the estimated disparity maps, it becomes clear that the few areas that are being removed by the post-filtering mostly arise in occluded areas and are thus legitimately removed, since it is not possible for the algorithm to reason about the depth in areas which are only seen by one camera.

Unfortunately, the results of the Middlebury 2014 stereo benchmark render the accuracy of our approach far from the state-of-the-art. With an error rate of over 35% for three out of four accuracy levels, the results are not really satisfying. Apart from the

high-resolution ground truth and the high accuracy levels in the evaluation, we assume that these poor results can also be attributed to the fact that the resolution, with which the disparity maps are being evaluated, is 4× bigger than the input resolution, introducing a lot of error in the process of upscaling and interpolation. Thus, we have also evaluated the results on ground truth data with only a quarter of the original image resolution and found that the error rate is on average reduced by way over 50%. For the configuration with a 5×5 census transform, being the best achieving configuration on this dataset, the accuracies calculated are 35.8% (bad0.5), 14.2% (bad1), 7.4% (bad2) and 4.9% (bad4), which is in the comparable range to that obtained on the KITTI benchmark.

As the name of our approach suggests, it is intended for real-time, rather than high-accuracy stereo processing. Our main use-case is the deployment on low-cost UAVs, equipped with embedded ARM or CUDA hardware, with the purpose of collision detection and avoidance or real-time 3D mapping and scene reconstruction. In the light of this use-case, we believe that the KITTI benchmark is more appropriate, since it comprises image data that depict real-world scenes in a quality that can also be expected from cameras mounted on UAVs. Moreover, for collision avoidance or real-time 3D mapping, it is not necessary to have disparity maps with high subpixel accuracy as evaluated by the Middlebury benchmark. It is more important to reliably detect the location of objects in the perceived scene and extensively reconstruct their appearance. Not only do the quantitative results prove the accuracy of the disparity maps, the qualitative presentation of some of the disparity maps also shows that our approach is able to reveal objects, which are only visible by a second glance, such as the person on the right of Figure 11 (row 2, column 2) or Figure 16 (column 2).

With respect to the KITTI 2015 benchmark, most of the configurations of our approach outperform the other approaches that perform real-time stereo estimation on embedded hardware. Even compared to the baseline algorithms, our results are compatible, especially when considering the frame rates we achieve. Furthermore, our experiments on the effects of subpixel disparity refinement (cf. Tables 4 and 5) have shown that its use can increase the accuracy by up to 35% without significantly decreasing the throughput (cf. Table 6). We therefore will consider using the post-refinement as part of the standard pipeline.

4.2. Processing Speed

Compared to the related work from literature, the throughput and processing speed of our approach are not very impressive. We expected to reach a significantly lower throughput than approaches running on FPGAs [17,18], as well as a slightly lower throughput compared to approaches that do not rely on a computationally expensive optimization scheme but run on an embedded GPU [26,41]. However, the $CT_{9\times7}$—SGM configuration of our approach has a lower throughput than that of a comparable configuration by Hernandez-Juarez et al. [25], while at the same time running on a hardware generation that is two times newer and that has twice the number of CUDA cores. This is not very satisfactory and something we will need to further investigate in the scope of future work. One difference between the work of Hernandez-Juarez et al. [25] and ours is that in our time measurements we include the data transfer to and from the memory of the GPU. Hernandez-Juarez et al. [25] argue that the processing can be overlapped with the computation and thus is not relevant for the computation of the throughput. However, in our case, the data transfer only takes up 4–6% of the processing time, which is not enough to reach the throughput of Hernandez-Juarez et al. [25], if omitted. Other optimization steps are the use of SIMD instructions to vectorize the cost aggregation with the CUDA kernels, or to streamline the aggregation of the last path and the disparity computation to reduce memory access, which should lead to a 1.35× performance speed-up [25]. A third and significant difference between our approach and that of Hernandez-Juarez et al. [25] with respect to performance is that Hernandez-Juarez et al. [25] refrain from any post-processing such as left–right consistency check or median filter. This allows the reaching of a higher

throughput but, in turn, reduces the accuracy of the results, leading to an error rate which is twice as high as ours in the actual estimate (cf. Table 2).

Furthermore, in the light of our use-case, addressing on-board stereo processing on low-cost UAVs, we believe that to this end the throughput of our approach is sufficient, as another limiting factor is the camera sensor and the data throughput between the sensor and processing board that is provided by commercial off-the-shelf (COTS) systems. Since a FPGA is typically located closer to the sensor with a direct and high-bandwidth connection, which allows the streaming of the image data directly into the memory of the FPGA, a high data throughput is of greater importance. However, embedded systems equipped with a GPU, such as the NVIDIA Jetson series that are mounted on COTS UAVs, are usually connected to the sensor via USB or similar, with the CPU capturing the data and storing it in global memory, from where it is transferred to the device memory of the GPU before it can be processed. This process does not allow for a high input frame rate (usually between 20 FPS and 30 FPS), as can be seen in the example of the DJI Matrix 210 (cf. Section 3.2), and therefore does not require an extremely high throughput. Nonetheless, the more time spent on the estimation of the disparity map, the less time is available for the successive interpretation, e.g., obstacle detection and avoidance. This raises our interest to further investigate the optimization of the processing speed in the future.

4.3. Power Consumption

The final aspect which we want to discuss is the power consumption of our approach running on the NVIDIA Jetson Xavier AGX in the light of deployment on a rotor-based UAV and whether it is feasible to use embedded CPUs for stereo processing. The average power consumption of the complete NVIDIA Jetson Xavier AGX (i.e., including the GPU, the CPU and the EMC) under the maximum power setting MAXN during the execution of our CUDA- and NEON-based approaches is approximately 20.1 W and 17.9 W, respectively. The DJI Matrix 210v2 RTK is powered by two batteries with a total energy of 349.2 Wh, which allows a maximum flight time of 33 min when no payload is attached [44]. Thus, during flight, the bare DJI Matrix 210v2 RTK consumes around 634.9 W per minute. This power consumption obviously increases with each gram of payload that is being attached. Given these measurements, we can calculate that the power consumption of the AGX under the highest power setting only makes up 3.2% and 2.8% with respect to the total power consumption of the DJI Matrix, thus reducing the flight time by a maximum of 1 min, when our CUDA- and NEON-based approaches are executed, respectively. This is an upper bound on the relative power consumption of the AGX, as the power consumption during flight of the DJI increases when payload is attached. The use of other power settings that will increase the FPS/W ratio (cf. Figure 14), but also decrease the absolute frame rate, is dependent on the use-case and whether the gain of a few seconds in flight time is more valuable than a major reduction in frame rate.

The power consumption during the execution of our CUDA-based approach is higher than during the execution of our NEON-based approach. This is expected as the GPU, which consumes more power than the CPU, is clocked down when not in use. However, the reduced power consumption does not stand in relation to the loss in processing speed of the NEON-based approach compared to the ones based on CUDA. This is clearly revealed by the curves in Figure 14, depicting that the NEON-based approaches running on the CPU have the worst FPS/W ratio. Thus, we conclude that the use of embedded GPUs is preferred over embedded CPUs. However, some drones are only equipped with an embedded ARM CPU, e.g., VOXL https://www.modalai.com/pages/voxl (accessed on 7 June 2021), for which a vectorized stereo processing with NEON intrinsics is an option.

5. Conclusions

In conclusion, we present an approach for real-time stereo processing on embedded ARM and CUDA devices, such as those attached to modern low-cost COTS UAVs. In this, we have optimized a disparity estimation algorithm for embedded CUDA GPUs,

such as the NVIDIA Tegra, by general-purpose computation on a GPU, as well as for embedded ARM CPUs by using the NEON intrinsics for vectorized SIMD processing. We have demonstrated that our approach reaches state-of-the-art accuracies when evaluated on public stereo benchmark datasets. Our CUDA-based implementation for stereo processing on embedded GPUs reaches real-time performance, even though it does not outperform related work in terms processing speed. The frame rates of our NEON-based implementation, however, outperform all related work on stereo processing on embedded CPUs. In a use-case specific scenario, we have further demonstrated the suitability of our approach for real-time stereo estimation on a low-cost COTS UAVs for the task of obstacle detection and 3D mapping by deploying it on a DJI Matrix 210v2 RTK equipped with a DJI Manifold 2-G.

We have shown that in the case of rotor-based UAVs a modern embedded GPU is a suitable alternative to an embedded FPGA, especially due to its shorter and thus less expensive developments cycles. Even though the GPU has a much greater power consumption than a FPGA and a significantly worse FPS/W ratio, its power consumption is negligible compared to the energy needed by rotor-based UAVs during flight and will reduce the flight time of the DJI Matrix 210v2 RTK by a maximum of 1 minute. However, for embedded systems with stricter power constraints, a FPGA-based approach should be considered. Our experiments have also shown that although the CPU requires less energy than the GPU, it has the worst FPS/W ratio. Thus, our optimization based on NEON intrinsics for vectorized SIMD processing should only be used if neither GPU nor FPGA are available.

Finally, we have also identified a few aspects to consider in future work. For one, we will need to further investigate which part of our optimization for CUDA-enabled GPUs can be further optimized, since our approach does not reach the processing speed of comparable approaches from the literature. And secondly, we should also consider other approaches, e.g., deep-learning-based algorithms that can reach higher accuracies than ours. Apart from that, our next steps are the extension of our approach by algorithms for real-time 3D mapping, as well as object and obstacle detection, to alleviate the perception of the environment around the UAV and in turn increase its autonomy.

Author Contributions: Conceptualization, B.R. and J.M.; methodology, B.R. and J.M.; investigation, B.R.; writing—original draft preparation, B.R. and J.M.; writing–review and editing, B.R. and M.W.; supervision, S.H. and J.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by the European Commission under the H2020 Framework Programme for Research and Innovation as part of the TULIPP project under grant agreement No 688403. We also acknowledge support by the KIT-Publication Fund of the Karlsruhe Institute of Technology.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Publicly available datasets were analyzed in this study. This data can be found here: http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo (accessed on 7 June 2021) and https://vision.middlebury.edu/stereo/eval3/ (accessed on 7 June 2021).

Acknowledgments: We would like to thank Raphael Senk and Shreyas Manjunath for their support in setting up and using the UAV with the on-board processing unit.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Stereoscopic Vision

It is well known that with the positions of two image points, p^L and p^R, in the images of the left and right camera of a calibrated stereo camera setup, belonging to the same scene point P, it is possible to compute the 3D position of P relative to the stereo rig. Thus,

finding such correspondences is the essential task in the process of stereo processing, and in turn dense disparity or depth estimation. Generally speaking, a correspondence search between an image point in the one camera and the image projection of the same scene point in the other camera must be done along the epipolar curve, which is the viewing ray, going through the image point of the first camera, projected into the image of the second camera. This correlation is formulated by the epipolar constraints. When the intrinsic parameters of the cameras are known, it is possible to account for image distortions, transforming the curve into an epipolar line. Furthermore, if the relative position and orientation between the two cameras of the stereo rig are known, the camera images can be rectified, i.e., they can be transformed onto a common image plane, and in turn the epipolar lines can be transformed to coincide with the image row of the image points. Thus, the difference between the image positions of p^{L} and p^{R} is reduced to a horizontal shift, the so-called disparity $d_{\mathrm{p}} = \left|p_x^{\mathrm{L}} - p_x^{\mathrm{R}}\right|$. This is illustrated in Figure A1.

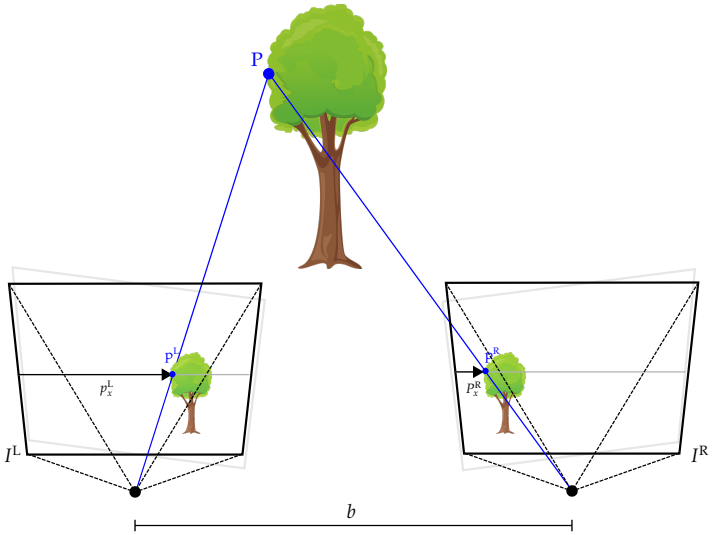


Figure A1. Illustration of a stereo camera setup. Two cameras, placed apart from each other with a distance b (baseline), observe a scene point P from two different vantage points. The scene point will appear at different locations, p^{L} and p^{R}, in the two camera images I^{L} and I^{R}. The difference between the x coordinate of the two image points, p_x^{L} and p_x^{R} is called the disparity d. If the camera rig is not calibrated, the two image planes are not rectified (light gray boxes), i.e., the image planes are not co-planar. By transforming I^{L} and I^{R} through rectification, the epipolar lines will align with the image row of the respective image points (gray lines).

Appendix B. Left–Right Consistency Check

The existence of occluded pixels that arise from areas, which are observed by one camera and yet occluded in the field-of-view of the other camera, is inherent to disparity and depth maps that are computed from a conventional stereo setup, consisting of only two cameras. A typical approach to detect and filter such pixels is the left–right consistency check. As illustrated in Figure A2, the disparities that are stored in the disparity map of the left image D^{L} are compared with the corresponding disparities in the right disparity map D^{R} and invalidated if they differ by a certain threshold, usually 1:

$$D(\mathrm{p}) = \begin{cases} D^{\mathrm{L}}(\mathrm{p}), \text{ if } \left|D^{\mathrm{L}}(\mathrm{p}) - D^{\mathrm{R}}(p_x - D^{\mathrm{L}}(\mathrm{p}), p_y)\right| \leq 1 \\ inv, \text{ otherwise.} \end{cases} \tag{A1}$$

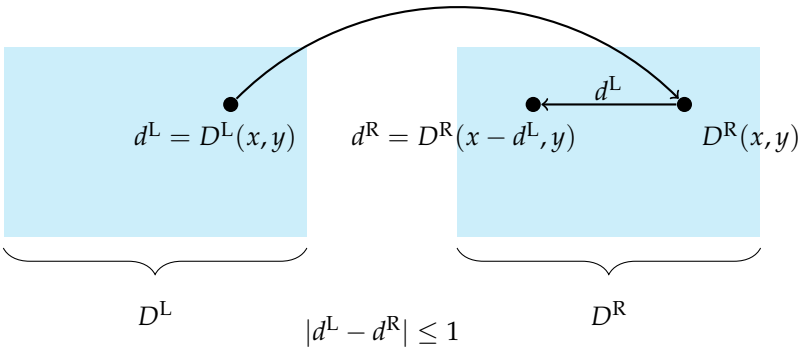


Figure A2. With the disparity maps D^{L} and D^{R} for the reference and matching image respectively, a consistency check according to Equation (A1) is performed to find occlusions.

Appendix C. General-Purpose Computing on CUDA-Enabled GPUs

Graphic processing units (GPUs) are designed for massively parallel processing. The number of processing units that are integrated into modern GPU hardware exceeds the number of cores available on a conventional CPU by far. Even embedded GPUs, such as the one built into the NVIDIA Jetson Xavier AGX, have up to 100× more cores than high-end desktop CPUs. However, the processing units on a GPU are less powerful and flexible than those of a CPU. Although the CPU is designed to do arbitrary processing tasks, the cores of the GPUs are intended for the parallel processing of small and dedicated instructions on a large amount of data simultaneously.

Massively parallel general-purpose processing on NVIDIA GPUs is alleviated by the CUDA-API, which allows the deployment of routines and functions for data processing in the form of so-called CUDA kernels on the GPU. When deployed, such a kernel is instantiated inside a high number of threads, which are being distributed among the available processing units and are each processing a different subset of the data. For a better abstraction and handling, the threads are logically grouped into thread-blocks, which share a local memory space and can thus exchange data between each other. Furthermore, the execution of all threads within one thread-block can be halted and synchronized. All thread-blocks are grouped into a grid. The grid size and the number of threads inside a grid is used to parameterize the instantiation of the kernel. The actual execution of the threads on a processing unit is always done in groups of 32, which is referred to as a CUDA warp. For an efficient general-purpose computation on a GPU (GPGPU), the developer is compelled to consider some design guidelines:

- Reduction of global memory access due to higher latency. Instead, data which is processed multiple times by threads in a thread-block should be cached inside the shared memory space.
- Pooling of global memory access and reduction of non-contiguous data storage.
- Efficient and maximum use of hardware resources.

Appendix D. Map Reduce Method

The map reduce method allows efficient discovery of the minimum of a given dataset. In this, each active thread performs a comparison between two elements and stores the smaller element in a designated memory space. The number of active threads, as well as the elements that are to be processed, is halved by each iteration as illustrated by Figure A3. Thus, the search for the minimum cost requires $\log_2(d_{\max})$ iterations. Ideally, the dimensions of the CUDA blocks are chosen in such a way that there are 32 active threads at the beginning. This allows that the map reduce algorithm can be processed in only one warp.

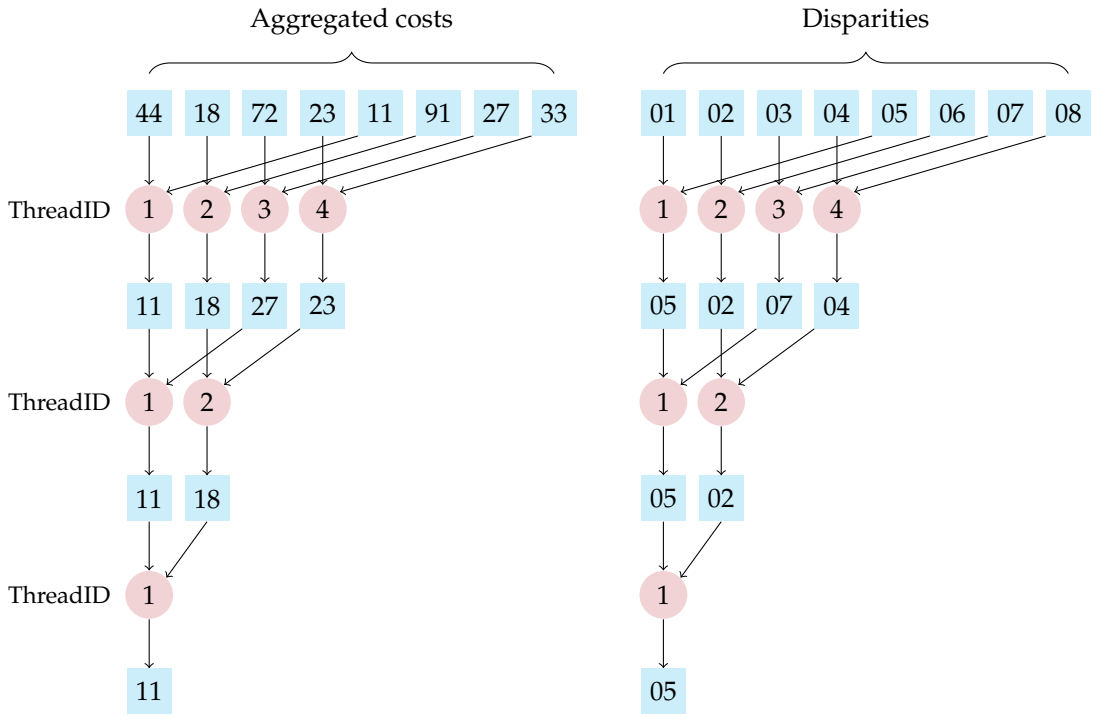


Figure A3. Illustration of the map reduce method to find the WTA disparity with the minimum aggregated cost. In each iteration, the number of aggregated costs that are being processed and the corresponding disparities are halved.

Appendix E. Vectorized SIMD Processing on ARM CPUs

Although conventional Single-Instruction-Single-Data (SISD) processing on the CPU requires one instruction for each data transfer between the memory and the registers of the CPU, as well as for each processing of the data stored inside these registers, the additional use of vector-processors allows the performance of one instruction on a set of data simultaneously, the so-called Single-Instruction-Multiple-Data (SIMD) processing. Each vector-processor is divided into multiple vector-lanes, which in turn hold one datum each. The memory unit as well as the arithmetic and logical unit of the vector-processor allow simultaneous transfer of multiple data into and from the lanes of the vector-registers, as well as to simultaneously combine the lanes of two registers and store the results into the lanes of a third register. The vector-processors of the ARMv8 architecture contain 32 vector-registers with a size of 128 bits each [45]. The number of vector-lanes inside each register differs, depending on the data type which is to be stored inside the register. Thus, each register can have 16 lanes when data with a size of 8 bits each is to be stored, or only two lanes, if each lane holds a datum with the size of 64 bits. In particular, image processing is well suited for the SIMD parallelization on vector-processors, since all pixels in an image are processed in the same manner, only with different data.

For an efficient use of vectorized SIMD processing, the developer is compelled to consider some design guidelines [32]:

- Reduction of the dependencies between conventional CPU and vectorized SIMD processing, to minimize the latency induced by copying data between the SISD and SIMD pipeline.
- Exploitation of cache coherence, to speed up the data transfer between the memory and the vector-registers.
- Dependencies in the data of vector-instructions trigger pipeline-stalls, in which the SIMD pipeline is stopped until the dependencies are resolved slowing down the processing.

- Minimal use of conditional branching, since if the Branch Prediction Unit (BPU) of the CPU predicts the wrong branch, the pipeline must be recursively cleared until the point of branching and restarted.

Appendix F. Sorting Networks

In general, sorting algorithms such as BubbleSort, MergeSort or QuickSort can sort an arbitrary number of input values and, in turn, require indefinite number of comparative and swapping operations, making a naive implementation inappropriate for vectorized processing. In the case of the fixed-size median filter, however, the number of considered input values are fixed and known in advance. This allows use of sorting networks [46], which sort an input vector of known size with a fixed number of comparators and swapping operations. A sorting network is comprised of two basic building blocks, namely:

- wires, which hold and transport one value of the input vector each, and
- comparators, which are responsible for comparing the values of the connected wires and swap these if necessary.

The comparators always connect two wires with each other and assign the smaller value to the upper wire. Figure A4 illustrates the BubbleSort algorithm implemented using a sorting network. In this, the horizontal lines represent the wires, while the comparators are illustrated by the vertical connections. The values of the input vector move along the wires from left to right and are rearranged by the comparators, resulting in a sorted output vector on the right with the smallest value at the top.

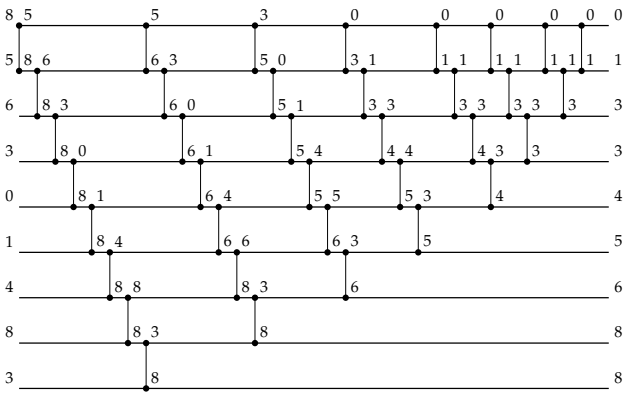


Figure A4. Exemplary sorting network, implementing the BubbleSort algorithm for nine input values.

References

1. Nex, F.; Remondino, F. UAV for 3D mapping applications: A review. *Appl. Geomat.* **2014**, *6*, 1–15. [CrossRef]
2. Restas, A. Drone applications for supporting disaster management. *World J. Eng. Technol.* **2015**, *03*, 316–321. [CrossRef]
3. Perz, R.; Wronowski, K. UAV application for precision agriculture. *Aircr. Eng. Aerosp. Technol.* **2019**, *91*, 257–263. [CrossRef]
4. Sebbane, Y.B. *Intelligent Autonomy of UAVs: Advanced Missions and Future Use*; CRC Press: Boca Raton, FL, USA, 2018.
5. Shakhatreh, H.; Sawalmeh, A.H.; Al-Fuqaha, A.; Dou, Z.; Almaita, E.; Khalil, I.; Othman, N.S.; Khreishah, A.; Guizani, M. Unmanned Aerial Vehicles (UAVs): A survey on civil applications and key research challenges. *IEEE Access* **2019**, *7*, 48572–48634. [CrossRef]
6. Nex, F.; Rinaudo, F. LiDAR or photogrammetry? Integration is the answer. *Eur. J. Remote Sens.* **2011**, *43*, 107–121. [CrossRef]
7. Hirschmueller, H. Accurate and efficient stereo processing by semi-global matching and mutual information. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Diego, CA, USA, 20–26 June 2005; pp. 807–814.
8. Hirschmueller, H. Stereo processing by semi-global matching and mutual information. *IEEE Trans. Pattern Anal. Mach. Intell.* **2008**, *30*, 328–341. [CrossRef] [PubMed]
9. Gehrig, S.K.; Eberli, F.; Meyer, T. A real-time low-power stereo vision engine using semi-global matching. In Proceedings of the International Conference on Computer Vision Systems, Liège, Belgium, 13–15 October 2009; pp. 134–143.

10. Banz, C.; Hesselbarth, S.; Flatt, H.; Blume, H.; Pirsch, P. Real-time stereo vision system using semi-global matching disparity estimation: Architecture and FPGA-implementation. In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, Samos, Greece, 19–22 July 2010; pp. 93–101.
11. Wang, W.; Yan, J.; Xu, N.; Wang, Y.; Hsu, F. Real-Time High-Quality Stereo Vision System in FPGA. *IEEE Trans. Circuits Syst. Video Technol.* **2015**, *25*, 1696–1708. [CrossRef]
12. Schmid, K.; Tomic, T.; Ruess, F.; Hirschmueller, H.; Suppa, M. Stereo vision based indoor/outdoor navigation for flying robots. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, 3–7 November 2013; pp. 3955–3962.
13. Gehrig, S.K.; Rabe, C. Real-time semi-global matching on the CPU. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, San Francisco, CA, USA, 13–18 June 2010; pp. 85–92.
14. Honegger, D.; Oleynikova, H.; Pollefeys, M. Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, 14–18 September 2014; pp. 4930–4935.
15. Barry, A.J.; Oleynikova, H.; Honegger, D.; Pollefeys, M.; Tedrake, R. FPGA vs. pushbroom stereo vision for MAVs. In Proceedings of the IROS Workshop on Vision-based Control and Navigation of Small Lightweight UAVs, Hamburg, Germany, 28 September–2 October 2015.
16. Hofmann, J.; Korinth, J.; Koch, A. A scalable high-performance hardware architecture for real-time stereo vision by semi-global matching. In Proceeding of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, Las Vegas, NV, USA, 26 June–1 July 2016; pp. 845–853.
17. Rahnama, O.; Cavalleri, T.; Golodetz, S.; Walker, S.; Torr, P. R3SGM: Real-time raster-respecting semi-global matching for power-constrained systems. In Proceedings of the IEEE International Conference on Field-Programmable Technology, Naha, Japan, 10–14 December 2018.
18. Zhao, J.; Liang, T.; Feng, L.; Ding, W.; Sinha, S.; Zhang, W.; Shen, S. FP-Stereo: Hardware-efficient stereo vision for embedded applications. In Proceedings of the IEEE International Conference on Field-Programmable Logic and Applications, Gothenburg, Sweden, 31 August–4 September 2020; pp. 269–276.
19. Ruf, B.; Monka, S.; Kollmann, M.; Grinberg, M. Real-time on-board obstacle avoidance for UAVs based on embedded stereo vision. *ISPRS Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.* **2018**, *XLII-1*, 363–370. [CrossRef]
20. Kalb, T.; Kalms, L.; Göhringer, D.; Pons, C.; Muddukrishna, A.; Jahre, M.; Ruf, B.; Schuchert, T.; Tchouchenkov, I.; Ehrenstråhle, C.; et al. Developing low-power image processing applications with the TULIPP reference platform instance. In *Hardware Accelerators in Data Centers*; Springer: Berlin/Heidelberg, Germany, 2019; pp. 181–197.
21. Rosenberg, I.D.; Davidson, P.L.; Muller, C.M.; Han, J.Y. Real-time stereo vision using semi-global matching on programmable graphics hardware. In *ACM SIGGRAPH 2006 Sketches*; Association for Computing Machinery: New York, NY, USA, 2006.
22. Ernst, I.; Hirschmüller, H. Mutual information based semi-global stereo matching on the GPU. In Proceedings of the International Symposium on Advances in Visual Computing, Las Vegas, NV, USA, 1–3 December 2008; pp. 228–239.
23. Banz, C.; Blume, H.; Pirsch, P. Real-time semi-global matching disparity estimation on the GPU. In Proceedings of the IEEE International Conference on Computer Vision Workshops, Barcelona, Spain, 6–13 November 2011; pp. 514–521.
24. Michael, M.; Salmen, J.; Stallkamp, J.; Schlipsing, M. Real-time stereo vision: Optimizing semi-global matching. In Proceedings of the IEEE Intelligent Vehicles Symposium, Gold Coast City, Australia, 23–26 June 2013; pp. 1197–1202.
25. Hernandez-Juarez, D.; Chacón, A.; Espinosa, A.; Vázquez, D.; Moure, J.C.; López, A.M. Embedded real-time stereo estimation via semi-global matching on the GPU. *Procedia Comput. Sci.* **2016**, *80*, 143–153. [CrossRef]
26. Chang, Q.; Zha, A.; Wang, W.; Liu, X.; Onishi, M.; Maruyama, T. Z2-ZNCC: ZigZag scanning based zero-means normalized cross correlation for fast and accurate stereo matching on embedded GPU. In Proceedings of the IEEE International Conference on Computer Design, Hartford, CT, USA, 18–21 October 2020.
27. Spangenberg, R.; Langner, T.; Adfeldt, S.; Rojas, R. Large scale semi-global matching on the CPU. In Proceedings of the IEEE Intelligent Vehicles Symposium, Dearborn, MI, USA, 8–11 June 2014; pp. 195–201.
28. Arndt, O.J.; Becker, D.; Banz, C.; Blume, H. Parallel implementation of real-time semi-global matching on embedded multi-core architectures. In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, 15–18 July 2013; pp. 56–63.
29. Rahnama, O.; Frost, D.; Miksik, O.; Torr, P.H. Real-Time dense stereo matching with ELAS on FPGA-accelerated embedded devices. *IEEE Robot. Autom. Lett.* **2018**, *3*, 2008–2015. [CrossRef]
30. Geiger, A.; Roser, M.; Urtasun, R. Efficient large-scale stereo matching. In *Proceedings of the Asian Conference on Computer Vision*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 25–38.
31. Saidi, T.E.; Khouas, A.; Amira, A. Accelerating stereo matching on mutlicore ARM platform. In Proceedings of the IEEE International Symposium on Circuits and Systems, Seville, Spain, 10–21 October 2020.
32. ARM. *NEON Programmers Guide*; ARM: Cambridge, UK, 2013.
33. Scharstein, D.; Szeliski, R. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vis.* **2002**, *47*, 7–42. [CrossRef]
34. Lowe, D.G. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vis.* **2004**, *60*, 91–110. [CrossRef]

35. Bay, H.; Tuytelaars, T.; Van Gool, L. SURF: Speeded up robust features. In Proceedings of the European Conference on Computer Vision, Graz, Austria, 7–13 May 2006; pp. 404–417.
36. Rublee, E.; Rabaud, V.; Konolige, K.; Bradski, G. ORB: An efficient alternative to SIFT or SURF. In Proceedings of the International Conference on Computer Vision, Barcelona, Spain, 6–13 November 2011; pp. 2564–2571.
37. Zabih, R.; Woodfill, J. Non-parametric local transforms for computing visual correspondence. In Proceedings of the European Conference on Computer Vision, Stockholm, Sweden, 2–6 May 1994; pp. 151–158.
38. Zhang, Z. A Flexible New Technique for Camera Calibration. *IEEE Trans. Pattern Anal. Mach. Intell.* **2000**, *22*, 1330–1334. [CrossRef]
39. Menze, M.; Geiger, A. Object scene flow for autonomous vehicles. In Proceedings of the Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 3061–3070.
40. Scharstein, D.; Hirschmüller, H.; Kitajima, Y.; Krathwohl, G.; Nešić, N.; Wang, X.; Westling, P. High-resolution stereo datasets with subpixel-accurate ground truth. In Proceedings of the German Conference on Pattern Recognition, Münster, Germany, 2–5 September 2014; pp. 31–42.
41. Cui, H.; Dahnoun, N. Real-time stereo vision implementation on Nvidia Jetson TX2. In Proceedings of the Mediterranean Conference on Embedded Computing, Budva, Montenegro, 10–14 June 2019; pp. 1–5.
42. Schönberger, J.L.; Sinha, S.N.; Pollefeys, M. Learning to fuse proposals from multiple scanline optimizations in semi-global matching. In Proceedings of the European Conference on Computer Vision, Munich, Germany, 8–14 September 2018; pp. 739–755.
43. DJI Onboard SDK Advanced Sensing-Stereo Camera. Available online: https://developer.dji.com/onboard-sdk/documentation/guides/component-guide-advanced-sensing-stereo-camera.html (accessed on 19 March 2021).
44. DJI. *Matrice 200 V2-Serie-User Manual*; DJI: Shenzhen, China, 2020.
45. ARM. *ARM Architecture Reference Manual for ARMv8 Architecture Profile*; ARM: Cambridge, UK, 2017.
46. Knuth, D. Sorting and searching. *Art Comput. Program.* **1998**, *3*, 513.