Balancing Accuracy, Speed and Power in the Computation of Depth with Stereo Cameras



Oscar Rahnama

Linacre College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2019

Acknowledgements

This journey would not have been possible without the help and support of many others. I would like to first thank my supervisor, Professor Philip H. S. Torr, who has always had my back and been there to support me throughout. Being a member of Torr Vision has been an honour and a privilege. I am also immensely grateful for having been a recipient of the Marie Skłodowska Curie ITN grant for project EDISON and I thank both Bhaskar Choubey and Martin Wäny for that opportunity. Likewise, I'd like to gratefully acknowledge FiveAI for their support in my final year.

Although the DPhil is an often solitary pursuit, having companions that you can rely on for help is invaluable. For that reason, I would like to particularly thank Tommaso Cavallari and Stuart Golodetz not only for their friendship, but for all the time, support and guidance that they have provided - having you onboard made a huge difference.

Having great folk alongside whom to spend your time on this planet makes day-to-day life altogether much more enjoyable. For that I am grateful to Deyan Levski, Duncan Frost, Guowei Tao, Rodrigo de Bem, Alessio Tonioni, Piotr Biliński, David Hansford, Mus'ab Shaharom, Alban Desmaison, Luca Bertinetto, Jishnu Mukhoti, Saumya Jetley, Anurag Arnab, Daniela Massiceti, Arnab Ghosh, Aravindh Mahendran, Heather Nelson, Ali Mohammadi, Nicholas Lord, Aleksej Makarov, Alessandro Brunetti, Francesco Giorgio, Nadia Pinton as well as many other present and past members of TVG, AVL, VGG, MCAD and FiveAI. I also want to thank the colleagues and friends I met whilst in Madeira, Portugal - you made me feel at home and had an enormous impact on my time there.

Thanks to all the friends and staff at Linacre College that have truly made me feel at home and part of the community. A particular thanks my close friends Robert Leśniak, Sangseraima Ujeed, Gregory McSweeney, Matthew Collins and Karolina Chocian for the unforgettable memories. An enormous thanks to my family who have always believed in and supported me in no matter what I do. I would not have gotten here without you. Lastly, I'd like to say a special thanks to my partner, Laura, for having always been there for me since the beginning - your love and support has meant the world to me.

Abstract

Many real-world applications in robotics and computer vision rely on the ability to acquire reliable 3-D information. Often, these systems also have strict requirements on the speed, latency, power consumption, form factor, usability and cost with which this capability is achieved. Although stereo cameras offer a cost-effective and versatile means of acquiring such depth information, using them involves computationally intensive processing over large amounts of data. At the same time, due to physical limitations on power consumption, the operating frequency of traditional single-core processors has plateaued. This has, consequently, spurred the adoption and development of multi-core and application-specific devices instead. As a result, algorithms can no longer be expected to simply improve in speed and efficiency with newer generations of processors. Instead, to achieve truly low-power and real-time systems requires algorithms that are purposely designed and optimized to account for the strengths and limitations of the underlying hardware.

Through efficient pipelining, extensive parallelism and conservative allocation of hardware resources, Field Programmable Gate Arrays (FPGA) are highly suited to fast, low-latency and power-efficient algorithm implementations. In this thesis, the capabilities of these devices are exploited for fast and power-efficient stereo image processing. Algorithms are investigated, developed and described for both strictly FPGA-only systems as well as heterogeneous CPU/FPGA System-on-Chip (SoC) platforms. The contributions of this thesis lie not only in the new approaches to low-power and real-time depth perception from stereo images that it proposes, but also in the insight that it provides regarding how to best utilize the FPGA-based devices for maximum performance. Simple, but highly parallel and predictable algorithms are shown to be great candidates for pure FPGA implementations, but more complex processing pipelines benefit more from the mixed computational strengths of the CPU-FPGA SoCs. The thesis also attempts to address the most significant disadvantage of using FPGAs, namely long development times involving highly specialized languages, by demonstrating how high-performing implementations can be obtained with the efficient use of high-level synthesis instead.

Contents

List of Figures vii

List of Tables ix

List of Abbreviations x

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Challenging Applications . . . 3
  - 1.1.2 Depth Sensing Approaches . . . 5
  - 1.1.3 Stereo-vision . . . 7
- 1.2 Author's Publications . . . 9
- 1.3 Open Source Projects and Patents . . . 10
- 1.4 Thesis Outline . . . 11

2 Background 12
- 2.1 The Transistor . . . 12
  - 2.1.1 Power Consumption . . . 13
- 2.2 Modern Computing Landscape . . . 16
  - 2.2.1 Central Processing Unit (CPU) . . . 17
  - 2.2.2 Graphics Processing Unit (GPU) . . . 19
  - 2.2.3 Field Programmable Gate Array (FPGA) . . . 20
  - 2.2.4 Application Specific Integrated Circuits (ASIC) . . . 22
- 2.3 The Choice of Device . . . 23

3 Fast and Strictly Local Stereo Image Processing 24
- 3.1 Preface . . . 25
- 3.2 Introduction . . . 25
  - 3.2.1 Depth From Stereo . . . 25
  - 3.2.2 Epipolar Constraint . . . 28
  - 3.2.3 Rectification . . . 29
  - 3.2.4 Related Work . . . 31
  - 3.2.5 Sum of Absolute Difference for Dense Disparity Maps . . . 32

3.3 Platform and Tools . . . 32
3.4 Architectural Considerations and Implementation . . . 33
3.4.1 Memory Management and SAD evaluations . . . 34
3.4.2 Minimum Evaluation . . . 37
3.4.3 Pipelining . . . 37
3.5 Results . . . 38
3.6 Discussion . . . 41
3.7 Conclusion . . . 42

4 Redeveloping "ELAS" for Speed and Power Efficiency 43
4.1 Preface . . . 44
4.2 Introduction . . . 44
4.3 Related work . . . 45
4.4 Preliminaries . . . 46
4.4.1 Original ELAS algorithm . . . 46
4.4.2 Platform . . . 47
4.4.3 High-level synthesis . . . 48
4.5 System Overview . . . 48
4.6 Key Accelerator Design Traits . . . 50
4.6.1 FPGA memory management . . . 50
4.6.2 Pipelining . . . 51
4.7 Platform Conscious Algorithm Changes . . . 52
4.7.1 Feature selection . . . 52
4.7.2 Measuring ambiguity . . . 56
4.7.3 Filtering support points . . . 56
4.7.4 Data Reduction . . . 57
4.8 Experimental Results . . . 58
4.8.1 Accuracy . . . 59
4.8.2 Per-frame processing time . . . 61
4.8.3 Power and resource consumption . . . 61
4.8.4 Throughput . . . 62
4.9 Comparison & Discussion . . . 63
4.10 Conclusion . . . 64

5 R^{3}SGM 66
5.1 Preface . . . 67
5.2 Introduction . . . 67
5.3 Background . . . 69
5.3.1 Semi-Global Matching (SGM) . . . 69
5.3.2 More Global Matching (MGM) . . . 71

5.4 Our Approach 72
5.4.1 FPGA Implementation 73
5.5 Experiments 76
5.6 Conclusion 81
5.7 Expansion to the Original Paper 83
5.7.1 Speed Bottleneck 83
5.7.2 Removing the Left Direction 84
5.7.3 Differences in Results 85
5.7.4 Summary 86

6 Combining Priors for More Accurate Real-Time Depth Processing 89
6.1 Preface 90
6.2 Introduction 91
6.3 Method 92
6.3.1 Sparse Disparity Computation 94
6.3.2 Combined Disparity Optimization 99
6.4 Results 101
6.5 Conclusions 102

7 Conclusion 103

References 107

List of Figures

1.1 Rabbit Stereogram 2
1.2 Examples of applications with challenging constraints 3
1.3 Man-made and natural methods of active distance measurement 5

2.1 Circuit depiction and illustration of N-type MOSFET (NMOS) 13
2.2 Dynamic power consumption within an CMOS Inverter 14
2.3 Simple CPU Block Diagram 17
2.4 The trend for microprocessors 18
2.5 General FPGA layout 20
2.6 Cost comparison of FPGAs, Structured ASICs and ASICs 22
2.7 Comparing CPU, GPU, FPGA & ASIC 23

3.1 Determining depth to an object with a two cameras 26
3.2 Distance w.r.t. camera pixel displacement, focal length and baseline . . 27
3.3 Epipolar Geometry 29
3.4 Image Rectification 30
3.5 Types of Radial distortion 31
3.6 Xilinx ZC706 development board 33
3.7 System-level depiction of SAD-based depth detector 34
3.8 FPGA memory management for efficient SAD computations 35
3.9 Pipelining within tree-based minimum locator 37
3.10 Example of minimum propagation through tree-based approach 38
3.11 Example depth images produced with SAD-based system on KITTI . . 40

4.1 ELAS algorithm overview 47
4.2 Grid-Vector extraction process 48
4.3 Overview of SoC adaptation of ELAS 49
4.4 FPGA memory requirements for efficient sliding window operations . . 51
4.5 Pixels involved in original ELAS features 53
4.6 Census Transform vs. Sum of Absolute Differences 55
4.7 One-dimensional simplification of redundancy verification 57
4.8 One-hot encoding of grid vectors 58
4.9 Processing time and related accuracy w.r.t down sampling 62

5.1 Comparing the regions of influence of SGM, MGM and R3SGM 69
5.2 Computing and storing unaries with rolling buffers 74
5.3 Rolling computation and storage of Census Transform feature vectors . 75
5.4 Rolling window buffers of disparity cost vectors 76
5.5 Qualitative examples of the approach on KITTI frames 79
5.6 Qualitative examples of the approach on Middlebury images 81
5.7 Isolating R^3SGM bottleneck 84
5.8 Impact of left direction on R^3SGM region of influence 85

6.1 System-level overview of heterogenous approach to depth computation 93
6.2 Comparing bias in raster and reverse raster passes of Fast R^3SGM . . . 94
6.3 Raster and Reverse Raster Passes 95
6.4 Example image of support points 97
6.5 Example of sparse anchor points image 98
6.6 Example of plane prior image 98
6.7 Example of final dense image 99

List of Tables

3.1 Impact of Varying Disparity range on Frame Rate and FPGA Usage. . . 39
3.2 Impact of Varying Input Image Resolution on Frame Rate and FPGA usage 39
3.3 Impact of Varying Window Size on Frame Rate and FPGA Usage. . . . 40
3.4 Comparing proposed evaluation against other similar implementations . 41

4.1 Impact of Window Sizes used during matching on frame rate, accuracy and resource utilization 60
4.2 Time Breakdowns 60

5.1 Comparison of accuracy of R3SGM on KITTI 2015 dataset 78
5.2 Accuracy of R3SGM compared to other FPGA-based approaches on Middlebury images 78
5.3 The frame rates achieved by R^3SGM in comparison to those achieved by other FPGA-based methods 80
5.4 Impact of varying Census Transform window size on FPS, error and resource utilization. 81
5.5 Results of 3-Direction R3SGM on KITTI 2015 86
5.6 Comparing the frame rate achieved of the modified version of R^3SGM against the original as well to the other FPGA-based methods. 87

6.1 Simple SAD vs. R^3SGM accuracy 90
6.2 Accuracy comparison of heterogenous FPGA approach to other state-of-the-art real-time methods on KITTI 2015 benchmark 100
6.3 Comparison of resource utilization and power consumption of heterogenous based approach 101

List of Abbreviations

1-D, 2-D	One- or two-dimensional, referring in this thesis to spatial dimensions in an image.
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CMOS	Complimentary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CT	Census Transform
DRAM	Dynamic Random-Access Memory
ELAS	Efficient Large-Scale Stereo Matching
FF	Flip flop
FPGA	Field Programmable Gate Array
FPS	Frames per second
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HDD	Hard Disk Drive
HDL	Hardware Description Language
LUT	Lookup table
MGM	More than Semi-Global Matching
MOSFET	Metal-Oxide-Semiconductor Field Effect Transistor
NMOS	N-type MOSFET
PMOS	P-type MOSFET
R^{3}SGM	Real-Time Raster Respecting Semi-Global Matching
SAD	Sum of Absolute Differences
SGM	Semi-Global Matching
SIMD	Single Instruction Multiple Data

SRAM . . . . . Static Random-Access Memory

SSD . . . . . Sum of Squared Differences or Solid State Drive

VHDL . . . . . VHSIC Hardware Description Language

VHLS, HLS . . Vivado High Level Synthesis

VHSIC . . . . . Very High Speed Itegrated Circuit

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

— Bertrand Russell

1 Introduction

Contents

1.1 Motivation 1
1.1.1 Challenging Applications 3
1.1.2 Depth Sensing Approaches 5
1.1.3 Stereo-vision 7
1.2 Author's Publications 9
1.3 Open Source Projects and Patents 10
1.4 Thesis Outline 11

1.1 Motivation

Consider the image in Figure 1.1 at the top of the next page: can you spot the rabbit? What at first glance appears to be a pattern of pebbles is something entirely different when viewed with stereoscopic vision. This ability of our brains to generate a three-dimensional illusion from two-dimensional images is not only incredible, but it also plays an essential role in our day-to-day existence. Its importance is captured best in a quote by Dr Paul Romano, a retired paediatric ophthalmologist who had temporarily lost sight in one eye following a massive ocular haemorrhage [1]:

> "I can't see spending the rest of my life in this way… Binocular stereoscopic depth perception is not just a visual phenomenon. It is a way of life… Life in a two-dimensional world is very different from that in a three-dimensional world and very inferior."



Figure 1.1: Chasing rabbits - can you spot the 3-D critter in the image? Staring at the image, relax your eyesight to beyond the page so that it appears blurry and doubled. Alternatively, bring the image close to your face, to the point where it is blurred and unfocused, then slowly move the image away whilst maintaining the defocus. Continue varying your focus in this way until a contour begins to appear in the middle of the image. Maintain that view and the rest of the details should then slowly come into focus to depict a 3-D image of a rabbit (as viewed from above).

Indeed, our visual sense in general, accounting for about 50% of the human cerebral cortex [2], is one of the richest sources of information that we possess about our surroundings. It is, however, our two forward-facing eyes which are central in allowing us to perceive depth and garner a three-dimensional sense of the world. Without them, we would be left to rely on visual cues to try and distinguish the relative positions and distances between objects. Such is the advantage of binocular vision that, across the entire animal kingdom, animals with the ability to see generally have, at the minimum, two eyes.

In addition to its incredible ability, the brain is also incredibly energy efficient. To get a sense of this, consider that an average human consumes about 2400 kilocalories per day to maintain their body weight. This is equivalent to $\sim$10,000 kJ over a 24-hour period and translates to an approximate continuous power consumption of 115 Watts. Of this 115 Watts, the human body dedicates the majority to thermoregulation to keep the body at 37.5 °C. Only about 20%, or 23 W, is used by the brain. Within this 23 W, the brain is able to consistently fulfil all of its functions, including seeing, hearing, feeling, learning, planning, remembering *etc*. Thus, matching the power consumption of a dim incandescent light bulb, the brain is an exceptionally power efficient information processor.

This seamless and power efficient 3-D perception of the world that we continually generate may seem natural and effortless to us, but providing the same ability to a machine is a challenging task. Nevertheless, depth perception plays an important role in many real-world systems such as robotic navigation [3–6], 3-D scene reconstruction [7–10],

object/human pose estimation [11–14], camera re-localization [15, 16], object detection [17, 18], manipulation and grasping [19] and human action recognition [20, 21]. Although every system that employs depth maps benefits from them being as accurate as possible, many also have additional requirements to consider. Resolution, acquisition speed, power consumption, financial cost, weight, shape, *etc.*, are all factors that can influence the choice of approach to depth perception.

1.1.1 Challenging Applications

The ultimate aim of this thesis is to provide suitable depth perception for applications that have particularly demanding constraints on speed, accuracy, power efficiency, form factor and versatility. Examples of such applications are illustrated in Figure 1.2 and can be generalized as mobile low-power real-time systems. Most of these systems share many of the following requirements:

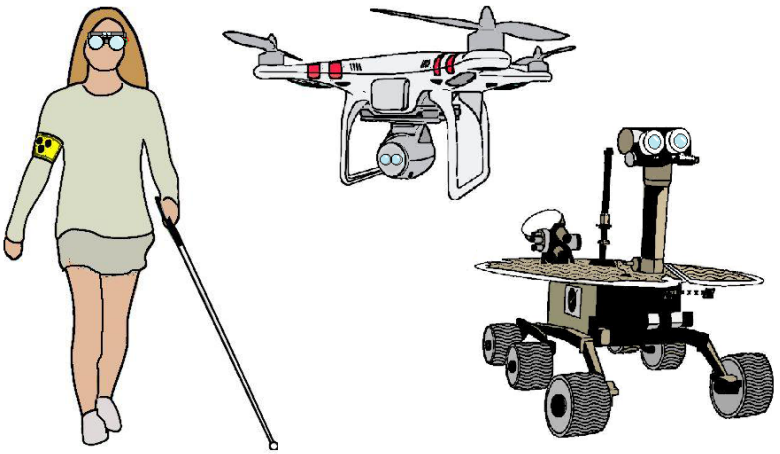


Figure 1.2: Example of outdoor systems requiring low-power, real-time depth perception. Prosthetic glasses, worn by the partially sighted, need to operate for long periods whilst being light and ergonomic. Small autonomous drones moving at high speeds require depth maps with low latency to avoid obstacles. Planetary rovers not only require power efficient electronics, but all of their onboard electronics must be robust to cosmic radiation. *Courtesy of Victor Rahnama.*

- **Real-time performance**
  Especially in safety-critical systems such as autonomous vehicles, a need exists for particularly reliable and fast acquisition of depth data [22]. More precisely, such systems must have access to data in "hard real time". Unlike "soft real time", where missing a deadline is permissible at the expense of poorer performance, "hard real-time" must unequivocally hit their deadlines or risk catastrophic failure. Small autonomous drones (Figure 1.2) that need to avoid obstacles and weave through

challenging environments (*e.g.* a forest) at high speeds also require this level of performance. On the other hand, prosthetic glasses benefit from high-speed depth to keep the user experience seamless, but the constraint can be relaxed to soft real time.

- **Low-Power Consumption**
  As these systems are inherently mobile, they have bounds on the energy with which to function. They either rely on limited amounts of energy harvesting, *e.g.* solar panels/small nuclear generators aboard planetary rovers, or operate on finite amounts of stored energy (fuel, batteries, *etc.*). Reducing power consumption therefore has a significant impact on their range or operational duration. A high power consumption can also generate large amounts of heat which then need to be adequately dissipated. Prosthetic glasses worn by the partially sighted, for example, must operate for hours at a time whilst running cool enough to avoid causing any discomfort to the wearer (Figure 1.2). Although the specific upper limits on power consumption vary depending on the application, generally, the lower the better.

- **Accuracy & Density**
  Most applications depend on the data to be sufficiently accurate to get a confident sense of the world around them. The degree of accuracy required, however, can vary. Often, it can be more important to avoid the occurrence of any single large mistake than it is to maximize the average precision of the measured distances. In these cases, the benefit obtained from improving precision may only be marginal. Confident but coarse approximations may, in reality, be sufficient. In the same way, most applications benefit from getting as much data as possible about the scene. As such, dense depth information, where roughly every point in the scene has an associated value, is often preferred to its sparse counterpart, where measurements are only provided for a subset of points. Again, in many applications, it is more important to have a general idea of the location of all points than it is to have more precise knowledge about a select few.

- **Challenging environments**
  All the applications depicted in Figure 1.2 require the ability to measure depth in a multitude of different and dynamic environments. This includes functioning in both indoor and outdoor environments and under various weather conditions. They should also not disrupt the environment in which they operate or interfere with themselves (*e.g.* prosthetic glasses used in public spaces). The semi-autonomous mars rover needs to additionally be robust to the hostile elements it is exposed to in space, most notably cosmic radiation.

- **Form factor and Weight**
  Due to the mobile nature of the systems in question, they often have strict requirements on form factor and weight. For autonomous aerial drones, this requirement is particularly strict due to the relationship between weight and lift. The heavier the drone is, the more power it requires to maintain the same amount of lift and, as a result, flight time and range are reduced. Attempting to compensate for this with additional battery capacity is inefficient as this itself further increases the weight of the drone. Moving or gyrating components should also be avoided on such platforms as they can affect manoeuvrability. Similarly, prosthetic glasses need to be sufficiently lightweight, inconspicuous and rigid to avoid discomfort to the wearer whilst remaining reasonably aesthetic. As for the planetary rover, launching objects into space is expensive and subject to the same fuel/propulsion dilemma that affects drones [23]. Consequently, any reduction in weight results in a greater payload capacity or a decrease in costs (due to a lesser need for rocket fuel).

- **Financial Cost**
  Lastly, the financial cost of a depth-sensing system can be an important factor to consider. Especially for systems intended for commercial sale, *e.g.* mass market prosthetic/augmented-reality glasses, keeping costs down plays a significant role in their economic viability.

1.1.2 Depth Sensing Approaches

One method of providing depth perception to a machine is through the use of active sensors. Much like echolocation in bats [24], these techniques involve emitting a signal

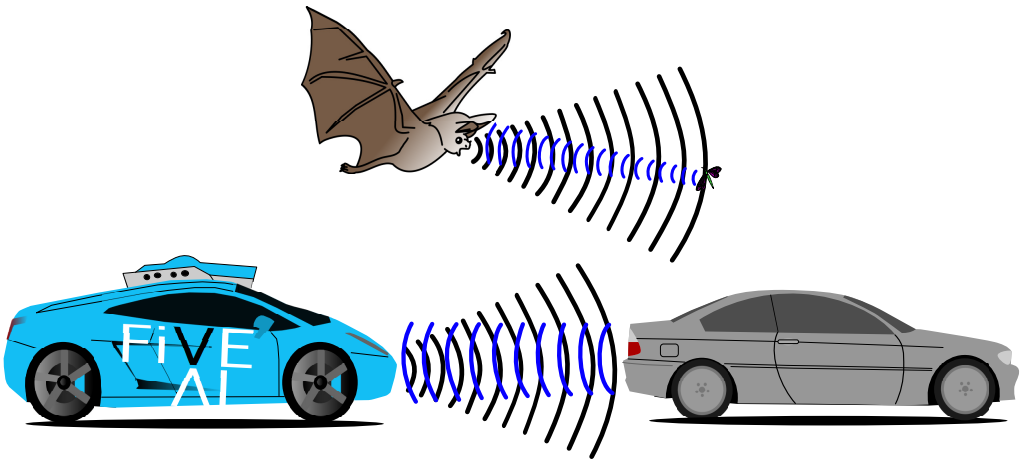


Figure 1.3: Active methods gauge distances in a similar fashion to bat echolocation. In both cases, a signal is emitted and the amount of time taken to receive a response is used to calculate distance to the reflecting object.

and measuring the time it takes for it to return as a means of determining distance. Approaches that fit into this category include ultrasonic sensing, Light Detection and Ranging (LIDAR), Radio Detection and Ranging (RADAR) and structured light sensing (e.g. Xbox Kinect V1) [25, 26]. These methods are accurate, fast and generally involve little processing, but also have many limitations. Most are limited in range as the strength of the reflected signal decreases with distance. Similarly, scattering or unwanted/multiple reflections can also affect the quality of the measurement. Those relying on infrared light, *e.g.* Time-of-Flight (ToF) or structured light, additionally struggle with ambient or environmental lighting, making them unsuitable outdoors where sunlight saturates their pixel sensors. On the other hand, capable of working outdoors, scanning LIDAR solutions such as Velodyne's HDL-64E [27] are bulky, expensive and only collect measurements for a sparse set of points. RADAR and ultrasonic sensors offer only limited resolution and can suffer from echoes. If many active sensors are operating in close proximity, they may also interfere with each other. Unlike bats which, in the presence of others, minimize interference through intermittent silence [24], safety-critical applications, such as autonomous driving, require systems that are persistently and continually operating with full reliability.

The other approach to obtaining depth, and the one developed in this thesis, is through passive sensing. Instead of *measuring* distance by actively polling for it, these methods *calculate* it from the information they passively collect through the use of cameras. Similar to human vision, the reliance on external sources of illumination means that they struggle in poorly lit conditions. Compared to active methods, however, they are inexpensive, lightweight, versatile (*i.e.* indoors/outdoors), low-power and can operate at long ranges. In fact, the distance to other stars in our universe was first accurately measured using the principles behind these methods [28]. As a result, passive methods are much more suited to the requirements of the mobile, low-power and real-time systems discussed in Section 1.1.1.

There are a many different ways of using cameras for passive sensing, but the two most common approaches are using a single camera (monocular) or using a set of two cameras (stereoscopic). Although recent approaches based on deep learning [29, 30] have made notable progress with monocular approaches, these struggle in determining scale [31]. Conversely, through the knowledge of the distance that separates the cameras (baseline), stereoscopic systems are not faced with the same issue and have absolute knowledge of scale. As a result, they are often more reliable and generally preferred. The greatest drawback of passive sensing, however, is the significant large and complex computational burden that it incurs to produce the required depth maps. In this way, the difficulty with stereo methods is not in the design of better sensors, but rather in the development of better algorithms.

1.1.3 Stereo-vision

As one of the oldest, most fundamental and widely studied problems in the field of computer vision [32–37], stereo matching continues to be one of the most active research areas, recently experiencing rising interest in deep learning based approaches [38, 39]. This has led to the development of many advanced and highly capable algorithms as evidenced by the impressive results listed at the top of popular stereo-vision benchmarks [35, 40–42]. In addition to providing a useful way of evaluating and comparing different methods to one another, these rankings help foster a competitive spirit within the field. With researchers continuously vying for the top spot on the leaderboards, a lot of the research effort is devoted to maximizing the benchmarks' metrics. Where this falls short, however, is that these rankings mainly only focus on the same single metric: Accuracy. As discussed in Section 1.1.1, this is not consistent with the requirements of many real-world applications for whom these algorithms are ultimately being developed. It is often equally important to consider power consumption, speed and latency. Although improving these characteristics has also been researched in the past [43–47], they have not garnered the same level of attention. Part of the reason behind this is that, unlike accuracy, speed and power efficiency cannot be considered in isolation. They are, instead, inextricably linked to the computing hardware that is used.

In the past, rapid progress and development in transistor technology meant that algorithms automatically ran faster and more efficiently with subsequent generation of computer chips. Eventually, however, the pace of this progress slowed as new difficulties with power consumption began to limit the speed of the classic single-core central processing unit (CPU). As a result, the focus then shifted towards developing chips with multiple cores; ultimately leading to today's multi-core processors and graphics processing units (GPU). These new chips offer greater computing power through their ability to operate on data in parallel. In many areas, such as deep learning, this has led to tremendous progress as previously unfeasible levels of computation can now be realized on these devices (particularly GPUs). However, not every algorithm is suited to benefit from this new parallel paradigm and only those that are will benefit from improvements in performance. Additionally, although high-performance CPUs and GPUs allow for a greater amount level of computational power, their very nature often means that they require vast amounts of power [48]. As an alternative to CPUs or GPUs, Application Specific Integrated Circuits (ASIC) and Field Programmable Gate Arrays (FPGA) can be used to design much more specific circuitry dedicated solely to processing a particular algorithm. The highly specialized nature of these approaches results in systems that are not only fast and low-latency, but which are also very power-efficient. However, it also means that they are more difficult to work with; often requiring more specialist

knowledge and longer development times. Although recent improvements in higher abstraction design tools have made using these types of devices more accessible and less time consuming, taking full advantage of them requires approaching algorithm development from the ground up with their strengths and limitations in mind.

This new need to co-design algorithms and hardware is beginning to be recognized by many practitioners in the field of computer vision as a whole [49]. No longer is it simply possible to rely on improvements in computing hardware to implicitly speed up an algorithm and reduce its power consumption for the large variety of applications where this is important. As described by Davison in his review of the future for application-oriented SLAM systems, the key to fast and power-efficient processing is to *divide computation between a large number of relatively low clock-rate or otherwise simple cores, and to minimise the movement of data between them* [49]. In many cases, it is often also recognized that using a heterogeneous mix of computing devices can best maximize the computational efficiency of a particular system.

Motivated by the pursuit of a solution that is suitable for mobile low-power real-time systems, this thesis is focused on developing stereo-vision algorithms that are optimally suited for deployment on FPGA-based platforms. Rather than solely prioritizing algorithm accuracy at the expense of all other attributes, it favours a balance between power consumption, speed and accuracy. By taking advantage of the parallel nature of stereo image data and the computational strengths of FPGAs, the resulting systems are feasible for deployment in the applications described in Section 1.1.1. The thesis builds off of previous work in the field of FPGA systems for stereo image processing and contributes several new findings and developments. Namely, it:

- Demonstrates the exceptionally fast frame rates that can be achieved on FPGAs with simple algorithms when in-stream processing is exploited to its fullest
- Finds new ways to adapt more complex algorithms onto FPGA-based platforms and highlights how their strengths can be taken advantage of
- Incorporates a CPU as part of an FPGA-CPU SoC to achieve more fine-grained and computation-specific acceleration
- Proposes new algorithms for stereo depth estimation that (i) maintain competitive accuracy scores whilst remaining simpler and more resource efficient than prior art (ii) make efficient re-use of hardware resource to allow for deeper processing pipelines where multiple passes across stereo image pairs provides improvements in accuracy and yet remaining fast and power-efficient.
- Provides regarding how to best utilize the FPGA-based devices for maximum performance

1.2 Author's Publications

Conference

- Alessio Tonioni, **Oscar Rahnama**, Thomas Joy, Luigi Di Stefano, Thalaiyasingam Ajanthan and Philip H. S. Torr, "Learning to Adapt for Stereo", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

- **Oscar Rahnama**, Tommaso Cavallari, Stuart Golodetz, Simon Walker and Philip H.S. Torr, "R^3GM: Real-time Raster-Respecting Semi-Global Matching for Power-Constrained Systems", in *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, Naha, 2018.

- **Oscar Rahnama**, Aleksej Makarov and Philip H.S. Torr, "Real-time depth processing for embedded platforms", in *Proceedings of SPIE 10223, Real-Time Image and Video Processing 2017*, May 2017.

- Aleksej Makarov, Vojislav Lukić and **Oscar Rahnama**, "Distance and speed measurements from monocular images", in *Proceedings of SPIE 9897, Real-Time Image and Video Processing 2016*, April 2016.

Journal

- **Oscar Rahnama**, Tommaso Cavallari, Stuart Golodetz, Alessio Tonioni, Thomas Joy, Luigi Di Stefano, Simon Walker and Philip H.S. Torr, "Real-Time Highly Accurate Dense Depth on a Power Budget using an FPGA-CPU Hybrid SoC", in *IEEE Transactions on Circuits and Systems II: Express Briefs (TCAS–II)*, vol. 66, no. 5, pp. 773–777, 2019.

- **Oscar Rahnama**, Duncan Frost, Ondrey Miksik and Philip H.S. Torr, "Real-Time Dense Stereo Matching With ELAS on FPGA-Accelerated Embedded Devices", in *IEEE Robotics and Automation Letters (RA-L)*, vol. 3, no. 3, pp. 2008-2015, 2018.

1.3 Open Source Projects and Patents

- The source code of the adapted and accelerated embedded CPU/FPGA implementation of the stereo matching algorithm presented in Chapter 4 is available online: `https://github.com/torrvision/ELAS_SoC`
- The work presented in Chapter 5 has been submitted for a "United Kingdom National Patent Application" with application number 1817390.6 and an "International (PCT) Patent Application" with application number PCT/EP2019/061426.
- The work presented in Chapter 6 has been submitted for a "United Kingdom National Patent Application" with application number 1817472.2.

1.4 Thesis Outline

The thesis's format is largely "integrated" with a subset of the papers listed on page 9, expanded upon and edited in places, accounting for the majority of the work presented. The remainder of the thesis is structured as follows:

- Chapter 2 serves as a background on the current state of computing technology. It discusses the main factors that affect power consumption and provides an overview of the different types of computing devices along with their respective strengths or weaknesses. By showcasing the strengths of FPGAs, it highlights their attractiveness to the application-oriented goal of real-time, low-power and accurate stereo-matching.

- Chapter 3 begins by describing the fundamental relationship between the distance to points in a scene and their disparity across pairs of stereo images. This includes covering some of the underlying principles of visual geometry that are used to simplify the matching process and compensate for the imperfect nature of real stereo-camera rigs. It subsequently develops, details and tests a simple, but remarkably fast FPGA-only stereo-matching system based on the Sum of Absolute Differences (SAD).

- In Chapter 4, the desire to obtain a more accurate algorithm leads to the development of an alternate approach where prior scene knowledge, extracted from an initial pass over the images, is used to influence a subsequent matching process. Building upon previously published work, this chapter examines the computational nature of the different stages of the algorithm to determine how to best adapt it for deployment on a hybrid CPU-FPGA platform.

- Chapter 5 draws inspiration from one of the most well-established stereo-matching algorithms to propose a new approach that is both similar in accuracy and better suited to real-time deployment on an FPGA where memory is limited. An expansion to this work is additionally included that discusses an alteration to this approach that allows for additional significant speedup.

- Chapter 6 builds upon the knowledge and experience gained from the work done in chapters 4 & 5 to present a novel hybrid CPU/FPGA pipeline that is capable of outputting dense depth maps with good accuracy. Despite its size, the entire pipeline is able to fit onto a single mid-range FPGA through efficient block re-use.

- Finally, in Chapter 7, a summary of the body of work is provided and possible future directions of research are discussed.

Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...

There is no one who loves pain itself, who seeks after it and wants to have it, simply because it is pain...

— Cicero's *de Finibus Bonorum et Malorum*

2 Background

Contents

2.1 The Transistor 12
2.1.1 Power Consumption 13
2.2 Modern Computing Landscape 16
2.2.1 Central Processing Unit (CPU) 17
2.2.2 Graphics Processing Unit (GPU) 19
2.2.3 Field Programmable Gate Array (FPGA) 20
2.2.4 Application Specific Integrated Circuits (ASIC) 22
2.3 The Choice of Device 23

This chapter serves mainly to provide context for the use of Field Programmable Gate Arrays (FPGA) as the basis for the research carried out throughout this thesis. It begins with an overview of the basic concepts behind modern computing machines then goes on to elaborate on the advantages and disadvantages between the different computing devices. Emphasis is placed on power consumption, processing throughput, versatility and cost as these make up the key sought-out criteria for real-world use as explained in the motivation. The goal of this chapter is to highlight the appeal of FPGAs, especially as they become more accessible with the progress of high-level synthesis tools, as platforms for low-power, real-time algorithm processing.

2.1 The Transistor

The most fundamental element in computer hardware is the transistor, and more specifically, the Metal-Oxide-Semiconductor Field Effect Transistor (MOSFET). Simply ex-

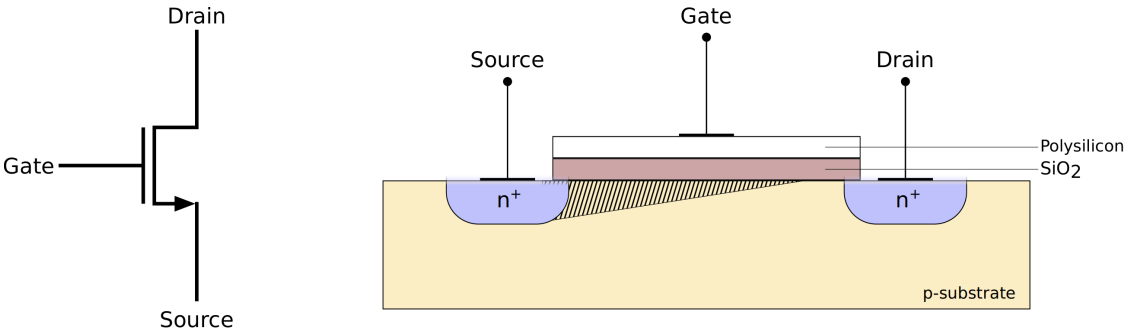


Figure 2.1: Left: Circuit Depiction of an N-type MOSFET (NMOS). Right: A simplified diagram depicting the different elements that make up an NMOS. A SiO_2 dielectric separates the polysilicon Gate (originally metal) from a positively doped (usually with boron) substrate. At both ends of the Gate, negatively doped regions in the silicon substrate (usually with Phosphorous) from the Source and Drain. The buildup of positive charge at the Gate creates an electric field that draws negative charges towards the complementary region in the p-type substrate. This accumulation of negative charge eventually forms a conductive channel through which current flows in the substrate.

plained, the transistor is a three terminal device that behaves like a voltage-controlled switch. In Figure 2.1, a typical N-type MOSFET (NMOS) is illustrated alongside its corresponding circuit diagram representation. In an NMOS, the presence of a sufficient positive voltage at the Gate causes the formation of a conductive channel in the underlying p-substrate; linking the Source and Drain and allowing current to flow. Active in this way, otherwise known as the transistor being in Saturation, the current that flows between the two terminals is largely a function of the voltage difference between the Gate and the Source and can be approximated as:

$$I_{\mathrm{D}} = \frac{\mu_n C_{\mathrm{ox}}}{2} \frac{W}{L} [V_{\mathrm{GS}} - V_{\mathrm{th}}]^2$$

where μ_n is the charge carrier's effective mobility, C_{ox} is the capacitance of the oxide layer, W and L are the width and length of the transistor, V_{GS} is the Gate to Source Voltage difference and V_{th} is the transistor's threshold voltage.

2.1.1 Power Consumption

The transistor is an active component, and therefore it consumes power. This power can be broken down into two distinct components:

$$P_{\mathrm{total}} = P_{\mathrm{dynamic}} + P_{\mathrm{static}}$$

The first, known as dynamic power, is the power drained by the transistor due to the activity involved in switching between active and non-active states. It is itself separated into two components,

$$P_{\mathrm{dynamic}} = P_{\mathrm{switching}} + P_{\mathrm{shortcircuit}}$$

Switching power, denoted $P_{\text{switching}}$, accounts for the majority of dynamic power consumption and stems from the continual charging and discharging of the inherent capacitance of the logic elements and interconnecting wires. Depicted on the left in Figure 2.2, the low (0) input to a CMOS inverter allows the current to flow from V_{DD} through the PMOS and into the output capacitance where it accumulates as charge and leads to a high (1) output. Conversely, when the input is switched to high (1), this previously stored charge is now drained through the NMOS transistor into Ground and therefore the output switches back to low (0). If a gate only seldomly switches states, then less power is used due to the decreased charging/discharging activity. The opposite is true when a gate is frequently switching between states and maximizing that activity. More formally, in a circuit, this type of dynamic power consumption can be calculated as

$$P_{\text{switching}} = \alpha C V_{DD}^2 f, \tag{2.1}$$

with C denoting the capacitance of the logic elements and interconnecting wires that are repeatedly charged to voltage V_{DD} with a clock frequency f and activity factor α. Short circuit power ($P_{\text{shortcircuit}}$), on the other hand, is due to the current drawn from a momentary short that occurs as a CMOS circuit switches between states. Depicted on the right in Figure 2.2, this short happens when both the NMOS and PMOS transistors are in transient states and simultaneously active for a brief moment as they switch from low to high or vice versa. As it only exists for a short instance, the amount of power that this accounts for is much less significant ($< 10\%$ of dynamic power) [50].

The second type of power dissipated through a transistor is static power consumption. This is the power consumption of the circuitry when the system is idle, even when the

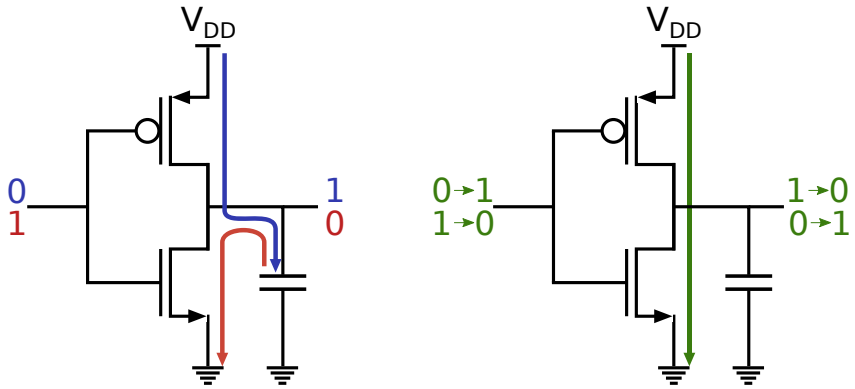


Figure 2.2: CMOS Inverter illustrating the two main components of dynamic power consumption. Left: Dynamic power due to switching activity which leads to the charging (blue) and discharging (red) of the logic and interconnect capacitance. Right: Dynamic power due to short-circuit current (green) occurring during transition between states where both PMOS and NMOS transistors are active.

transistors are thought to be "off". This happens because transistors are not ideal devices and therefore there is always some *leakage current* that flows between the different terminals of the devices. There are different types of leakage current, with subthreshold leakage being the most dominant. As implied by the name, subthreshold current is present when the voltage at the Gate is below the threshold voltage, V_{th}. This current flows between the Source and Drain of the transistor and is at its peak when the Gate voltage is close to V_{th} [50]. More broadly, the combination of all the sources of leakage currents can be denoted as I_{DD} and the power drained as a result simplified to:

$$P_{\text{static}} = I_{DD} V_{DD} \tag{2.2}$$

The Power Wall

Ever since MOSFETs were first invented, engineers have been relentlessly working to improve them. The rate of this development in transistor technology was famously observed by Gordon Moore in 1965 - director of R&D at Fairchild Semiconductor at the time (later becoming cofounder/CEO of Intel) - as a doubling in the number of transistors every year [51]. This trend, revised in 1975 to a doubling of the transistor count every two years, is commonly referred to as *Moore's Law* [52]. In some sense a self-fulfilling prophecy, *Moore's Law* spurred the decrease in process size from the early 50μm nodes to the 7 nm ones currently available from the leading foundries (Intel, Samsung, TSMC). Not only has this miniaturization allowed for much greater transistor density, but it has also enabled faster switching frequencies and decreased the overall power consumption of individual transistors [52].

To understand this relationship, one must consider how the shrinking process size affects the charging and discharging of the voltage at the Gate. Behaving like a capacitor (Figure 2.2), the Gate voltage of a MOSFET is governed by

$$V_C = V_s(1 - \mathrm{e}^{1-\frac{t}{RC}}). \tag{2.3}$$

The charging time is controlled by the time constant RC where the value of capacitance C is related to the size of the Gate and the thickness of the SiO_2 dielectric with

$$C = \varepsilon_0 \frac{A}{d} \tag{2.4}$$

As transistors are made smaller, both the width and length that make up the area A of the Gates shrink and, as a result, capacitance C decreases. Although this is somewhat offset by the thinning of the SiO_2 dielectric d, the comparatively larger and quadratic scaling down of A dominates in impact. Referring to Equation 2.3, a smaller capacitance implies a smaller time constant RC and therefore the Gate charges and discharges more

rapidly (faster transistor switching frequency). Also, as reflected in Equation 2.1, this decreases the dynamic power consumption. More intuitively, less capacitance means that the Gate has less capacity to store charge and therefore it reaches saturation both quicker and with less charge drawn. In addition, while the decrease in thickness of the SiO_2 layer only has a small effect on capacitance, it has a significant impact on lowering dynamic power consumption by decreasing the threshold voltage V_{th}. This reduction in V_{th} allows the use of a smaller voltage V_{DD}) which, as denoted in Equation 2.1, has the largest impact on $P_{\text{switching}}$.

Although the miniaturization of transistor technology has had a significant positive effect on dynamic power consumption P_{dynamic} , it has also had the opposite effect on static power consumption P_{static} . With transistor sizes now within an order of magnitude of silicon atoms (235 pm) and with the terminals getting ever smaller and ever closer together, leakage current has increased. Most notably, although the thinning of the dielectric benefits $P_{\text{switching}}$ by lowering V_{DD}, the related decrease in V_{th} has the added effect of increasing subthreshold leakage current [50, 53]. As a result, where once engineers focused solely on reducing dynamic power and paid little to no attention to static power, the power lost due to leakage current has now become an equally important consideration [54–56].

This has left chip designers confronted with a "power wall", where they cannot make transistors any faster. In essence, they are no longer able to compensate for the additional power needed by an increase in f through a decrease in V_{DD} (Equation 2.1) as this would now lead to a surge in leakage power instead [57]. Without the ability to adequately cool the chips and dissipate that extra heat, the maximum transistor frequency has plateaued. To keep pushing for additional computational power, chip designers have now taken a fundamentally different approach. Rather than pushing transistor technology to the limit for more speed, throughput is now increased through additional parallelism. Importantly, however, this means that one can no longer rely on future chips being faster and more power-efficient to improve the performance of an algorithm - shifting to a parallel paradigm is the only remaining option [57].

2.2 Modern Computing Landscape

The shift towards parallelism means that the classic central processing unit is no longer the most suitable device for every type of process. The modern computing landscape is now abundant with a variety of computing machines that each has advantages and drawbacks depending on the application. By providing an overview of the most prominent types of devices, this section provides the rationale behind the choice of the field-programmable gate array to achieve the goals set out in this thesis.

2.2.1 Central Processing Unit (CPU)

The cornerstone of modern day computing, the general-purpose central processing unit (CPU) is at the heart of most computers. At its simplest, depicted in Figure 2.3, it consists primarily of an arithmetic logic unit (ALU), registers, an instruction fetcher/decoder and a memory controller to access external memory. The ALU, capable of integer arithmetic and bit-wise logic operations, is fed decoded instructions fetched from memory to operate on the data kept in its registers. If the necessary data is not in the local registers, the CPU goes through the memory controller to get it from global memory. As fetching data from the DRAM or the disk drives (HDD/SSD) is time consuming, modern CPUs generally also have multiple levels of low-latency SRAM, called cache, which maintain quicker access to the most recently or repeatedly requested data. Although this adds complexity by needing to ensure coherency between the different levels of memory, it significantly helps to minimize the number of dead cycles the CPU spends waiting for data. Nevertheless, as the data access pattern cannot always be predicted ahead time, the CPU's performance can be severely hampered by frequent accesses to longer term memory.

The classic single-threaded CPU is a strictly sequential machine that can only process one operation at a time. The time taken to process a set of instructions is therefore governed by

$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}} \tag{2.5}$$

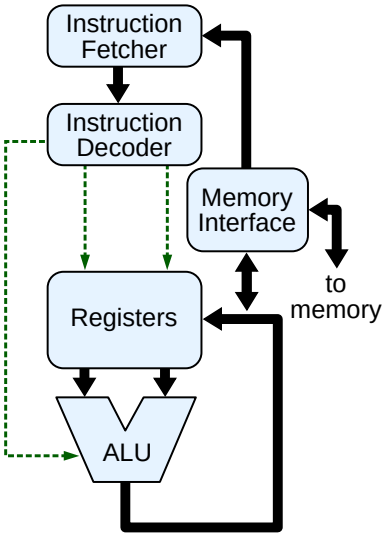


Figure 2.3: Simple CPU Block Diagram. Instructions, fetched and decoded from an instruction stack, that direct the ALU to either: 1) Operate on the data stored in the registers 2) Load or Store data between global memory and the registers.

or more succinctly,

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock Frequency}} \tag{2.6}$$

where CPI is clock cycles per instruction. Its overall throughput can be increased by clever pipelining and prediction, but it ultimately only processes one instruction per clock cycle. Not being able to decrease the instruction count or CPI, the only way to speed up a CPU process is through a faster clock frequency. As discussed in Section 2.1.1, though, clock frequency can no longer be increased without significantly impacting the power consumption of the chip. This is further amplified by the fact that, for every watt of energy dissipated, an additional 0.5 W to 1 W of power is needed to cool the device [58, 59]. For highly sequential processes with very dynamic and random accesses to memory, the CPU can be the ideal computing device due to its high clock frequency. However, the combination of the strictly sequential modus operandi, the volatile memory access times and the inability to increase speed without significantly dissipating more power makes the CPU a poor choice for power-conscious real time and inherently parallel algorithms.

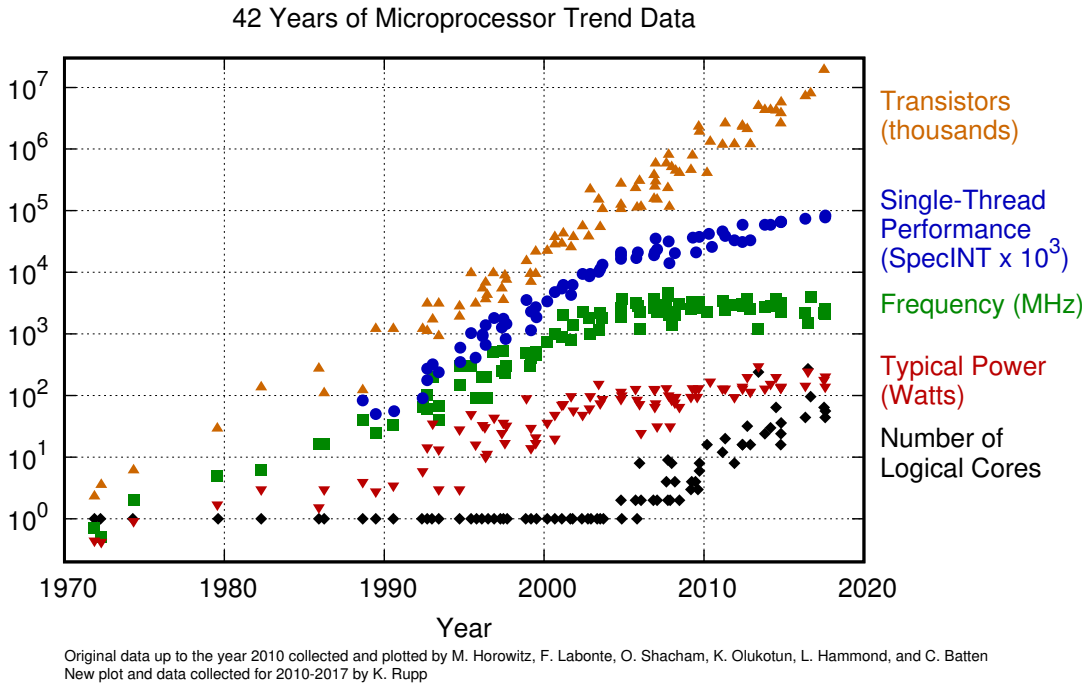


Figure 2.4: The trend for microprocessors. With engineers reaching the limits of individual transistor switching frequencies, chip designers have shifted their focus towards manufacturing CPUs with multiple cores.

In an attempt to introduce more parallelism and increase compute without affecting power consumption, chip designers began to develop CPUs with multiple cores. As shown in Figure 2.4, the new norm has become to cap the maximum frequency of single

cores but to have additional cores as part of the same CPU. As a result, the number of processes that can be run in parallel on different threads is much greater. This is best suited when the task can be separated in such a way that the different threads minimize the amount of data they share. If multiple threads require the same data, competition for access as well as the need to ensure coherent caches can lead to many dead cycles across the cores. This stalling of threads is inefficient and significantly reduces performance.

2.2.2 Graphics Processing Unit (GPU)

Traditionally used alongside the CPU with the primary purpose of real-time graphics rendering, the graphics processing unit (GPU) - and more precisely, the general-purpose GPU (GPGPU) - has become an important and formidable source of computational power [60]. At the basic level, a GPU is made up of an enormous number of individual cores which can range from hundreds to thousands. Like with multi-core CPUs, this multitude of cores allows for many threads to be executed in parallel and thus increased computational throughput. Unlike CPUs, though, these threads are not entirely independent of another. Instead, the GPU uses a "single instruction, multiple data" (SIMD) framework where multiple cores execute the same instruction across different pieces of data. Accordingly, it excels when the computations are predictable and repetitive, but loses the advantage when the datapath contains a lot of branching [61]. In such a situation, the GPU is inefficient as it must allocate enough resource to account for each possible outcome of the branching. Similarly, it struggles if there is a lot of inter-thread data dependence as the need to maintain data coherence keeps cores idle and waiting.

The combination of high clock frequency (albeit slower than CPUs) and vast number of cores makes GPUs power intensive devices; with the largest and most capable devices dissipating as much as 300 W. Despite this, they are typically more computationally power efficient than CPUs as the difference in power is overshadowed by the difference in performance [62–68]. They nonetheless suffer from inefficiencies due to underutilization of resources [69] and memory-bandwidth-related performance bottlenecks that cause cores to idle [70, 71]. Furthermore, although low-power GPUs are available (*e.g.* Nvidia TX1/TX2 < 10 W), their smaller size and memory can oblige them to communicate with the CPU more frequently. This creates additional overhead which severely degrades performance [72, 73].

The adoption of GPUs has been aided and accelerated by the ongoing development of higher-level methods of programming them (*e.g.* CUDA, OpenCL) and by the ease with which the processes are automatically distributed and scheduled by the device itself. Although this has made GPUs more accessible to non-specialists, it also means that programmers do not generally have as much fine-grained control over how and when

different tasks are processed. As a result, despite the general speedup, one is often left to rely on the worst-case execution to get an idea of reliable timings. In addition, although GPUs have high computational throughput, they have longer latency due to the time needed to initially shuffle data onto local memory and appropriately distribute it among the cores. For these reasons, the GPU is not ideal for real-time applications that require minimal latency and strict guarantees on execution times.

2.2.3 Field Programmable Gate Array (FPGA)

The field-programmable gate array (FPGA) is a distinctly different device to both the CPU and GPU. Rather than being a general-purpose processor, the FPGA is a type of general-purpose "(re-)programmable hardware". It does not have any complex ALU cores, memory hierarchies or instruction sets and is instead simply a large matrix of interconnected logic blocks - a sort of digital "play-doh". At the low level, the basic logic element that makes up an FPGA is a lookup table (LUT) followed by a latch/flip-flop. By linking and combining these units together, the FPGA establishes a semi-permanent circuit that is solely dedicated to handling the processing of a given algorithm. To increase performance, modern chips often include additional blocks to simplify the design by making certain operations more efficient. This includes components dedicated to common operations such as addition, functional units dedicated to digital signal processing (DSP slices) as well as memory blocks used to more efficiently store and manage data (*e.g.* Block RAM, UltraRAM).

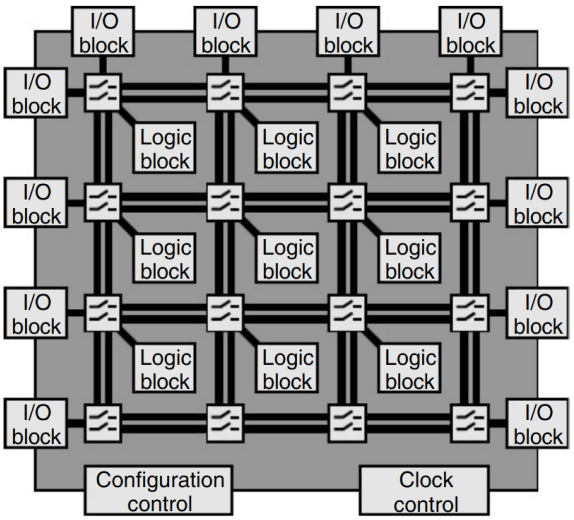


Figure 2.5: General FPGA layout
Image taken from Bailey 2012 [74]

By only utilizing as many logic blocks as needed to implement a given function and by virtue of their slower clock frequency (Equation 2.1), FPGAs are very resource

and power efficient devices. Indeed, although they do have inefficiencies due to their generic and re-programmable nature, FPGAs are often at least an order of magnitude more power efficient than GPUs [62–64, 75–79]. This power efficiency does also not come at the cost of performance, with FPGAs equalling [80, 81], if not surpassing [63, 68], the capability of GPUs. They are particularly well suited to algorithms that can be computed in a streaming manner where the data access pattern is regular and predictable. Their computational advantage stems from the highly deterministic nature of the circuit they implement. This, along with the large amount of resources at their disposal, allows FPGAs to process many different computations in parallel whilst simultaneously ensuring high throughput through optimal pipelining of successive operations. Their deterministic nature also makes them very suitable to hard real-time applications as their latency and execution times are known to an exact number of clock cycles. However, like GPUs, FPGAs are not very well suited to algorithms with unpredictable/dynamic memory accesses or a lot of inherent branching and forking. The lack of cache memory means that unforeseen requests of data from external memory introduce significant delays and degrade performance. On the other hand, branching is handled by implementing every possible outcome into the device and therefore leads to increased resource usage and logic complexity. This affects both power efficiency and performance.

Hardware Description and High-Level Synthesis

Another drawback of FPGAs is that, compared to both CPUs and GPUs, they are relatively difficult to use. Historically, using FPGAs required the use of specific hardware description languages (HDL) such as VHDL or Verilog. Not only do these languages have low levels of abstraction, but they differ substantially in style from traditional programming languages. The process is more of a description than it is a list of commands and its synchronous nature adds additional complexity, *e.g.* consecutive statements executed simultaneously instead of sequentially. As a result, traditional FPGA development is often long, tedious and only done by hardware engineers and specialists.

Recently, alternative programming methods have been developed to help make FPGAs more accessible. These methods offer higher-level abstraction by either featuring a framework such as OpenCL or a "C-to-Gates" style approach. C-to-Gates approaches involve writing code in a higher-level language that is then compiled and synthesized into FPGA code. A good example is Xilinx's Vivado High-Level Synthesis (VHLS) that uses C/C++. Using such a tool benefits the programmer by abstracting away many of the tedious details found in traditional HDLs and thus frees them to direct more effort towards developing the core functionality. In addition, the higher-level code is also easier to reuse and understand by non-specialists. Nevertheless, using these tools effectively still

requires the programmer to possess a strong understanding of the underlying hardware. The burden of profiling and then accurately "describing" the algorithm as a system of hardware elements - rather than a sequence - is still very much on the programmer.

2.2.4 Application Specific Integrated Circuits (ASIC)

An Application Specific Integrated Circuit, or ASIC, is a highly specialized chip dedicated to a single purpose. Out of all the approaches described in this section, an algorithm implemented as an ASIC will yield by far the fastest and most power-efficient performance. Compared to an equivalent FPGA implementation, the ASIC would occupy 20-40 times less silicon, be 3-4 times faster and consume 10-15 times less dynamic power [82]. Of all the options, it is also the least flexible and the most expensive. Such a device requires a team of engineers, working over an extended period, to design and manufacture. Also, once fabricated, ASICs cannot be modified - any changes require repeating this lengthy and costly process. As shown in Figure 2.6, ASICs are generally only ever considered when high volumes are needed. This is because, although the initial non-recurring engineering costs are large (*i.e.* designing the masks), once successfully produced, the individual unit costs are small.

An alternative option that acts as an intermediate between FPGAs and ASICs is the *Structured* ASIC. Like an FPGA, the structured ASIC contains a sea of prefabricated gates that it can synthesize into a circuit. It does so, however, permanently - through the addition of a number of metal layers. In essence, although more expensive than an FPGA, it outdoes it in both efficiency and performance for a fraction of the cost and development time of a standard ASIC approach. As the metal masks can be reused, as shown in Figure 2.6, it is most cost-efficient for mid-volume level designs.

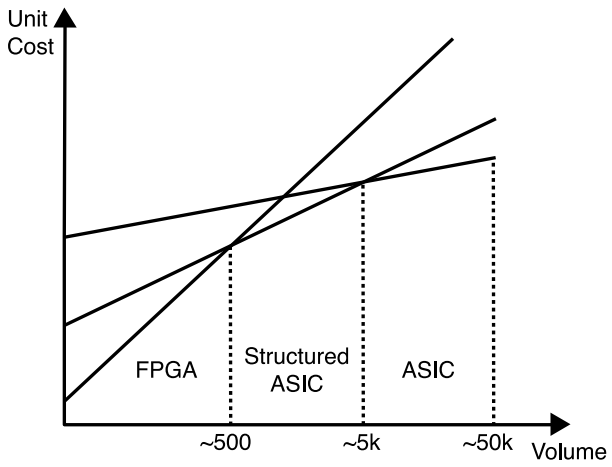


Figure 2.6: The cost of using FPGAs, Structured ASICs or ASICs with volume of units produced.

2.3 The Choice of Device

Considering the tradeoffs between the different computational platforms (Figure 2.7), the most suitable option is the one that best balances the different requirements discussed in Section 1.1.1. A strictly CPU approach cannot provide the necessary throughput as it is unable to take full advantage of the innate parallelism of the algorithms. Conversely, although it would offer the best performance and power efficiency, the enormous cost and time to develop an ASIC makes it an unattractive choice. In addition, both the ASIC and Structured ASIC lack sufficient flexibility - leaving little room for experimentation or improvement. Between the GPU and FPGA, the former has the benefit of being easier to adopt and program, but the latter offers significantly better power efficiency and is more apt at providing guarantees on execution times. In particular, FPGAs have also separately been shown to be better suited for stereo, optical flow and other similar sliding-window type algorithms [83, 84]. As achieving low-power and real-time ability are two of the main criteria that are being pursued, these factors clearly favour the FPGA.

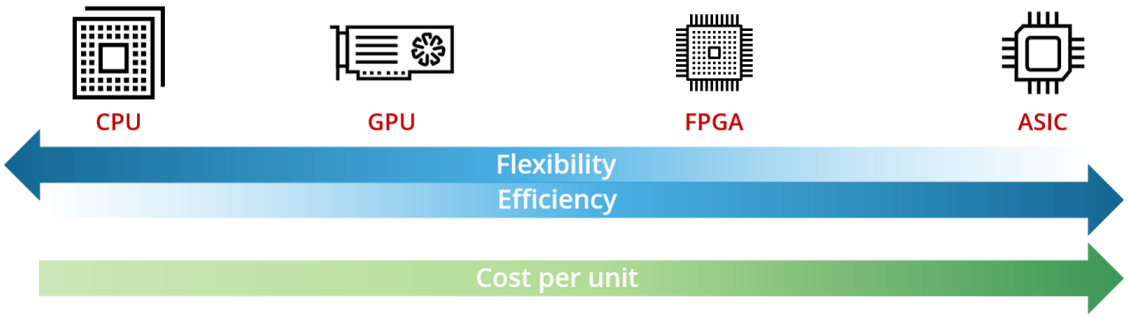


Figure 2.7: Tradeoff in cost, flexibility and efficiency between CPUs, GPUs, FPGAs and ASICs.

Recently, FPGA manufacturers have started to offer devices that combine both and FPGA and a low-power ARM CPU on the same silicon chip. These new CPU-FPGA system on chips (SoC) enable much faster and more efficient data communication between the two components. As a result, the complementary strengths of both devices can be effectively leveraged to accelerate algorithms that involve a mix of both highly parallel and highly dynamic computations. Compounded by the recent availability of higher abstraction approaches of programming FPGAs, the choice of FPGA and the CPU/FPGA SoC as platforms for mobile low-power and real-time depth from stereo is well justified.

In order to know virtue, we must first acquaint ourselves with vice.

— Marquis de SAD

3

Fast and Strictly Local Stereo Image Processing

Contents

3.1 Preface 25
3.2 Introduction 25
3.2.1 Depth From Stereo 25
3.2.2 Epipolar Constraint 28
3.2.3 Rectification 29
3.2.4 Related Work 31
3.2.5 Sum of Absolute Difference for Dense Disparity Maps 32
3.3 Platform and Tools 32
3.4 Architectural Considerations and Implementation 33
3.4.1 Memory Management and SAD evaluations 34
3.4.2 Minimum Evaluation 37
3.4.3 Pipelining 37
3.5 Results 38
3.6 Discussion 41
3.7 Conclusion 42

3.1 Preface

This chapter reflects the work published in "Real-time depth processing for embedded platforms" [85] with myself as the primary author. Compared to the original paper, the contents of this chapter are mostly identical, but there are a few exceptions for the purposes of completeness and clarity. Most notably, the section providing an overview of the computation of depth from stereo cameras has been expanded to include some of the preliminary notions and assumptions, such as those obtained from epipolar geometry, that are made in the context of this image processing task. An additional table has been added to the results to better compare the method to previous work. The introduction of the paper has also been shortened, as it would otherwise have repeated much of what was already stated in the introduction of the thesis.

3.2 Introduction

In this chapter, an embedded implementation of a pure Sum of Absolute Differences (SAD) approach to stereo matching is described. This simple local algorithm is widely used as a fundamental element in many more accurate and complex techniques [86]. The objective is to maximize the throughput of this basic computation in order to set an upper bound on the speed achievable with more complex methods. In addition, it aims to minimize power consumption of the system making it suitable for mobile/embedded applications. Compared to GPUs, FPGAs are more power efficient and are better suited to sliding window operations such as the SAD [83, 87]. Importantly, the design of the hardware is approached from a higher level of abstraction to retain competitive development times compared to GPU programming.

The chapter begins by providing an overview on computing depth from stereo cameras as well as a description of the SAD algorithm. In the following section, it describes both the embedded platform on which the SAD algorithm is implemented and the tools that are used. In Section 3.4, the architecture of the SAD hardware block is described in detail and the techniques used to achieve high throughput are explained. Results are presented in Section 3.5 and their implications discussed in Section 3.6.

3.2.1 Depth From Stereo

To calculate a depth image, a stereoscopic camera system exploits the differences between the two images it captures, together with the knowledge of how each image relates to the other. The underlying principle can be simulated by fixing your gaze on a very distant point whilst having one of your fingers in the middle of your field of view. When

alternating between the views from each one of your eyes, the finger's apparent position changes significantly between the views whilst that of the distant point remains relatively unchanged. This observed difference in images provided by a stereoscopic system is that which is measured to calculate depth maps. It is important to note that whilst the distance between a camera and an object is a quantity that depends on external factors, such as the camera's mounting height, tilt, pan and skew angles, depth is a measure of the distance between the observed points in space and the camera's image plane. Depth images can also be "dense" or "sparse", with the former denoting that there is an associated depth value for roughly every pixel in the image and the latter denoting depth images where only a subset of the pixels have associated values. The explanation of how to obtain the value for depth beneath begins with the assumption of an ideal stereoscopic rig. Although no system is ideal in the real world, images can be rectified to correct for this - as will be shown later.

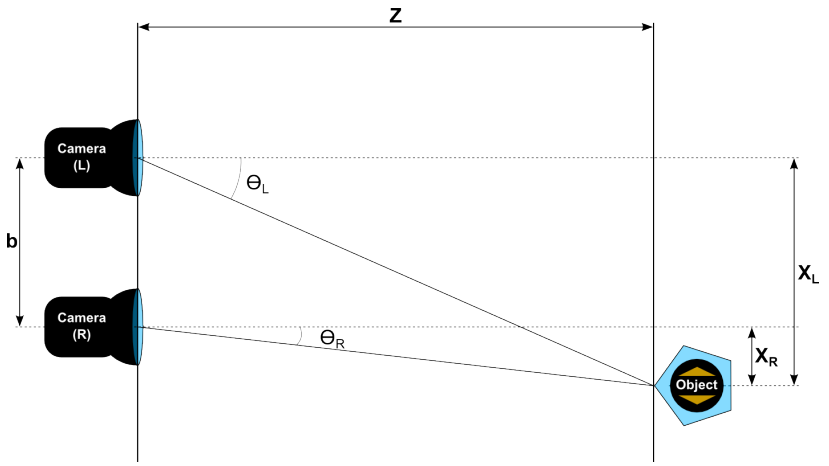


Figure 3.1: The depth z of an object is a distance between it and the image plane. It can be computed from the baseline (the distance between the sensors) and angles under which the object is seen from each of the sensors with respect to the optical axis.

In Figure 3.1 , two identical cameras, referred to as L (left) and R (right) cameras, with identical sensors and lenses $(fL = fR = f)$ are parallel, spaced from each other by a baseline b and at depth z from a point on an object. From Figure 3.1, the depth z can be obtained using trigonometry:

$$z = \frac{b}{\tan(\theta_L) - \tan(\theta_R)} \tag{3.1}$$

From this alone, however, neither the value of angles θ_L and θ_R nor their corresponding horizontal displacements x_L and x_R are obtainable. Referring to Figure 3.2, the angles θ_L and θ_R can instead be obtained from the focal length f and horizontal displacement

of the projected point across the two images by triangle similarity. Therefore, the depth z can be re-expressed as

$$z = f\frac{b}{d_0}, \tag{3.2}$$

where d_0 is metric disparity, i.e. the metric distance between pixel positions (columns) in the left and the right frames.

Since the separation is measured over image data, this disparity is calculated as a pixel column separation and needs to be transformed with:

$$d_0 = (C_R - C_L)\frac{w_s}{N_c} \tag{3.3}$$

where w_s is the sensor width, N_c is the total number of columns per sensor, while C_L and C_R are the columns in the left and right frame onto which a feature of the object is projected. Combining Equation 3.2 and Equation 3.3, depth is inversely proportional to disparity, and to its scaled version $C_R - C_L$,

$$z = \frac{f}{w_s}N_c\frac{b}{C_R - C_L}. \tag{3.4}$$

Equation 3.4 describes how to measure distance from the difference in pixel location of a point projected onto two coplanar, horizontally aligned images. This assumes, however, that the location of the projected point within both images is already known. In reality, the

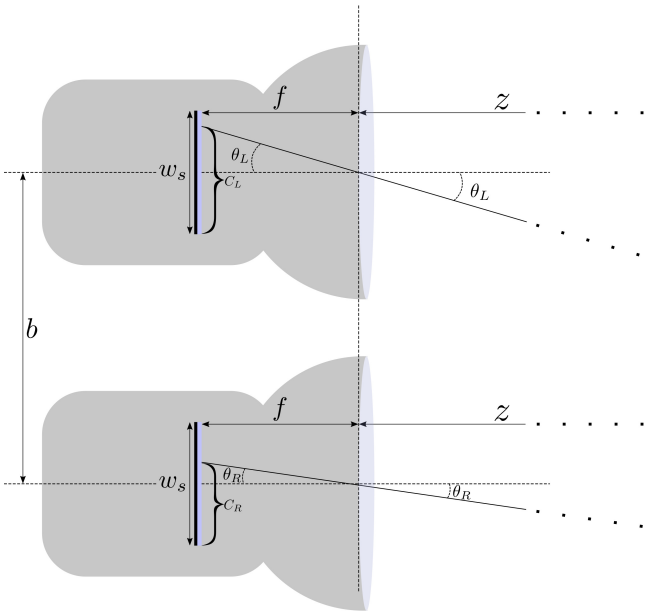


Figure 3.2: The depth z of an object computed from its projections onto pixels in columns C_L and C_R of horizontally aligned, identical left and right sensors. Each sensor has width w_s, focal length f and a total number of pixel columns N_c.

process of finding these correspondences is wherein lay the most difficulty, even without considering occlusions. Naively, one could begin by comparing pixels from one image against all of the pixels in the other image. Thanks to the epipolar geometry, this search space can be significantly reduced to a single line in the image called the *epipolar line*. Furthermore, as stereoscopic systems are non-ideal systems in real-world applications, the matrix relating points and corresponding search lines, referred to as the *fundamental matrix*, can be leveraged to rectify and calibrate stereo images. This remaps the images and ensures that the epipolar lines over which matches are considered are horizontal - simplifying the data access pattern and computational complexity of the matching process.

3.2.2 Epipolar Constraint

To understand how the searching process can be simplified, the basics of epipolar geometry need to first be explained. A stereoscopic system is depicted in Figure 3.3 with the left and right cameras having their optical centres at O_L and O_R respectively. The origin of the 3D world reference system is set to O_L and O_R differs from it by a translation t and a rotation R. The baseline connecting the two camera centres, $O_L O_R$, intersects both image planes at their respective epipoles, e_L and e_R. These epipoles are the projections of each camera's optical centre into the other camera's image plane. With both cameras recording a point P as p_L and p_R in their respective image planes, the *epipolar plane* is the plane connecting all of these points. Notably, in stereo matching, where one point in one image is known and its correspondence is being searched for in the other image, it is the knowledge that both are coplanar along this plane that limits the search space. This *epipolar constraint* limits the search space to the intersection of the epipolar plane and the camera image plane, also known as the *epipolar line*.

In an intuitive sense, the epipolar line ℓ_R is the 2-D projection onto the right camera's image plane of the ray that passes through both P and optical centre O_L. It is therefore certain that somewhere along this projection of the ray, the projection of the point itself, p_R, also exists. In this way, assuming a point is not occluded and is contained within the field of view of both cameras, a mapping exists between its projection in one image and a line in the other image where its other projection can be found.

Epipolar and Fundamental Matrices

This mapping is described by the *essential matrix* $E \in \mathbb{R}^{3\times3}$, originally introduced by Longuet-Higgins [88] in 1981. A matrix of rank 2, E captures the geometric relationship between two cameras within the normalized coordinate system. To establish this more

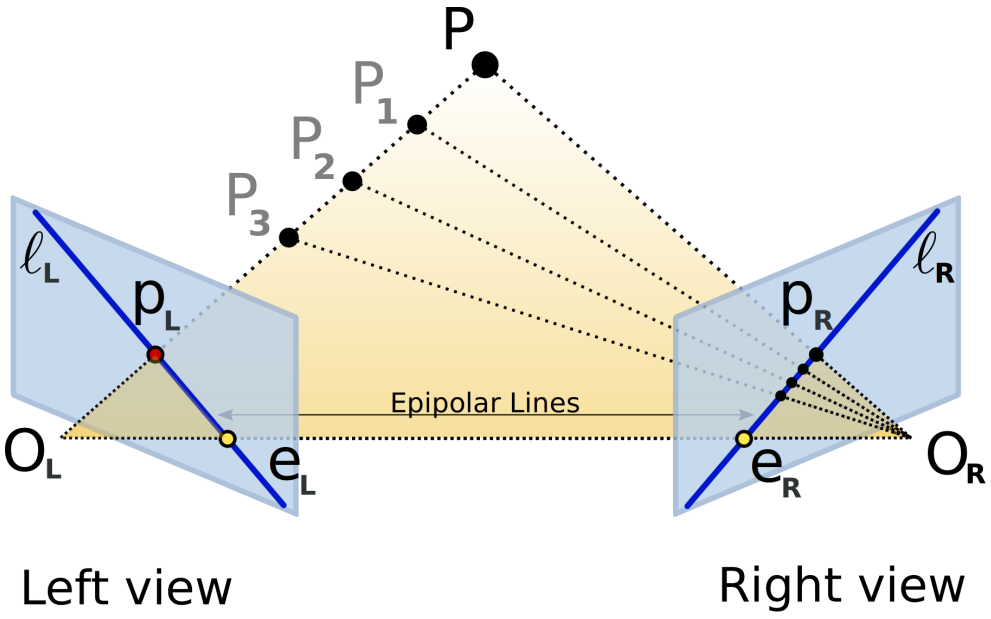


Figure 3.3: Epipolar Geometry

directly with camera image coordinates, the more general *fundamental matrix* F exists and relates to E with

$$F = {K_R}^{-\top} E K_L^{-1}$$

where K_R and K_L are the intrinsic camera matrices of the right and left camera respectively. In this way, F maps $p_L \rightarrow \ell_R$ and satisfies the condition that for any pair of corresponding points $\mathbf{x} \leftrightarrow \mathbf{x}'$

$$\mathbf{x}'^{\top} F \mathbf{x} = 0$$

F has seven degrees of freedom, two in addition to the five contained in E, and can be estimated by the "eight-point algorithm" introduced by Hartley [89].

3.2.3 Rectification

In the ideal system depicted in Figure 3.1, both cameras are horizontally aligned and level. The epipoles in this type of system are located at both infinities along the x-axis, and as a result, all the epipolar lines along both image planes are horizontal. These horizontal epipolar lines are particularly useful when searching for correspondences, as not only is the search space confined to a 1-D line, but they are also confined to lay along the same row of pixels in the images, *i.e.* scan line. This allows for much more efficient memory access and, as will be discussed later, necessary to allow for efficient and fast computations on FPGA platforms. It is therefore particularly useful that, given

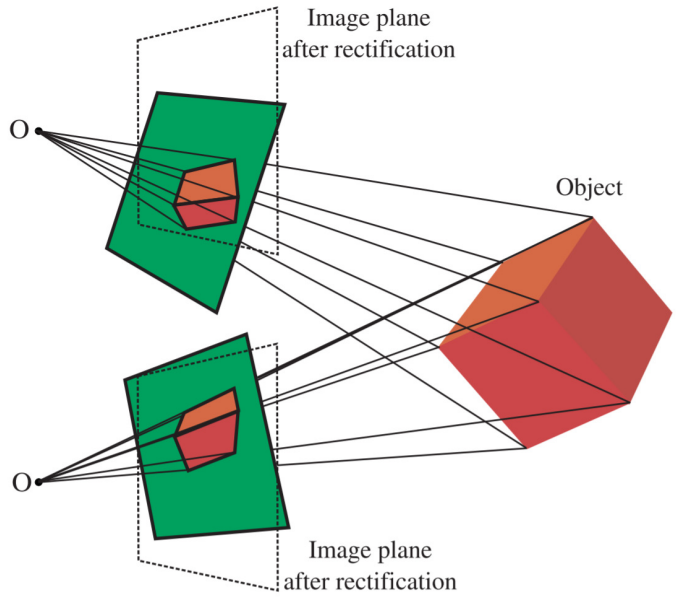


Figure 3.4: Through a process of homography, images captured from a non-ideal stereo camera setup can be rectified so that they lay on the same plane with their epipolar lines horizontally aligned. This ensures that correspondences are found along the same scan line. *Figure taken from [92].*

the fundamental matrix F, it is possible to warp the images from any non-ideal stereo system to appear as though as they were captured by an ideal, horizontally aligned system. This process of horizontal *rectification*, depicted in Figure 3.4, is achieved through a homography of the images onto the same plane [90–92].

Furthermore, unlike an ideal pinhole camera, real cameras use lenses to focus the incoming light onto their image sensors. These lenses deform the images with both radial and tangential distortion [93]. Radial distortion, either "barrel" or "pincushion" as illustrated in Figure 3.5, is symmetric about the centre of the image and has the effect of bending straight lines. Tangential distortion, on the other hand, is a result of the misalignment of the lens with the image sensor where the two are not parallel. Techniques exist to both characterize and correct these distortions so that the resulting images are "dewarped" [93–95].

Both the lens dewarping and the horizontal alignment of the epipolar lines play an important role in the rectification of the images for the purposes of finding correspondences in depth maps. They can generally be established ahead of time through a process of calibration and then efficiently implemented through the use of lookup tables. For the work contained herein, the assumption is made that the images have already been rectified. This is also how test images from established stereo matching benchmarks are generally provided.

3.2.4 Related Work

Typical implementations reported in literature are based on Sums of Squared Differences, Sums of Absolute Differences and the Census Transform. Dense matching algorithms can be categorized as local or global based on whether they rely only on local intensities within a widow or whether they include smoothness constraints by penalizing neighbouring pixels' different disparities. Local dense matching is computationally more efficient though more prone to errors in the absence of textures [96],[97]. While local methods compute the matching cost for every pixel, a more global inference and better accuracy are usually achieved by cost aggregation over a window of a fixed size [98], variable size [99, 100], or size range [101].

It has been shown [102] that the real-time performance on resource-limited systems, *i.e.* the capability of processing 30 frames per second on an FPGA, DSP or ASIC, is in most cases achieved by implementing Sums of Absolute Differences (SAD). Similarly, Kalarot et al. [103] have shown that despite a much slower internal clock, FPGA implementation of the (SAD-based) Symmetric Dynamic Programming stereo vision algorithm (SDPS) is 30% faster than its corresponding GPU implementation. This was achieved through pipelining: no waiting time was imposed for the aggregation of intermediary results; they were immediately passed to succeeding pipeline stages. It is also worth noting that, of the eighteen fastest algorithm implementations on FPGAs [102], more than half (ten) were based on the sum of absolute differences (SAD).

To seek an upper bound to the achievable throughput of a disparity estimating system, the pure SAD is preferred due to its widespread adoption as an essential component of other, more complex, techniques. The reason being that, as a basic component of other algorithms, a high frame rate implementation of the SAD sets the upper bound of that which is achievable with more complex techniques.

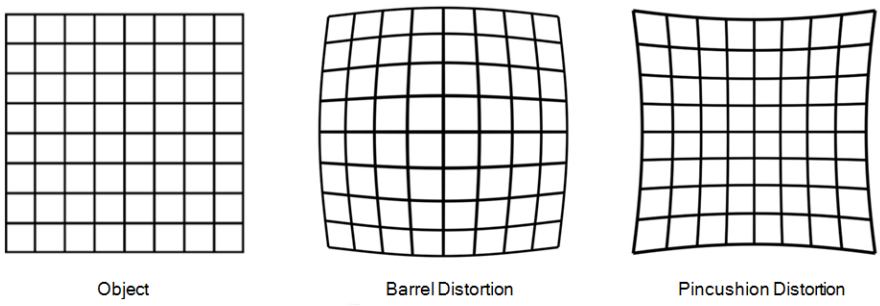


Figure 3.5: Types of Radial distortion. Barrel distortion, commonly observed with fish eye lenses, is noted when the image magnification decreases as the distance from the centre of the image increases. Conversely, pincushion distortion squeezes the image, increasingly magnifying pixels the further they are from the centre.

3.2.5 Sum of Absolute Difference for Dense Disparity Maps

For its simplicity and versatility, the Sum of Absolute Differences has been used in a number of computer vision applications, ranging from motion estimation and image compression [104], target matching and object tracking [105], through image stabilization [106] to adaptive optics [107]. When estimating depth of an object at a particular location (r,c) in the left image, a matching algorithm should typically compare image intensity for a certain range of disparities of interest. Assuming the 3D world reference system has its origin in the centre of the left camera image, the sum of absolute differences can be expressed as:

$$SAD(r,c,d^{(i)})_{L-R} = \sum_{m=-w}^{w}\sum_{n=-w}^{w}\left|I_L(r+m,c+n) - I_R(r+m,c-d^{(i)}+n)\right|, \tag{3.5}$$

where r and c denote the row and the column number of the pixel at which disparity is evaluated, the aggregation area (window) size is $(2w+1)\times(2w+1)$ and d denotes a disparity assumption expressed in pixels,

$$d^{(i)} = c_R^{(i)} - c. \tag{3.6}$$

The index (i) denotes the i$^{\text{th}}$ iteration, because the sum of absolute differences is computed for a range of disparities $d^{(i)} = \{0,1,2,\ldots,d_{max}\}$. Once the sum of absolute differences has been computed for every disparity candidate, the best match disparity is computed as:

$$D(r,c) = arg\,min_i\Big(SAD(r,c,d^{(i)})\Big). \tag{3.7}$$

3.3 Platform and Tools

The system is implemented on the Xilinx ZC706 development board which incorporates their Zynq-7045 All Programmable System-on-Chip [108]. Depicted in Figure 3.6, this is a heterogeneous platform containing programmable logic and a discrete processing system. The programmable logic portion is the equivalent of a Xilinx's Kintex-7 series stand-alone FPGA and contains 350K programmable logic cells. A programmable logic cell is a basic processing unit of FPGA consisting of a logical function generator in form of a lookup table (LUT), a D-type flip-flop (a binary storage element of shift registers), a couple of multiplexers and a connection to adjacent cells. Logic cells in Kintex-7 and Zynq-7000 series are organized in slices and configurable logic blocks (CLB), where each CLB consists of two slices and each slice contains 4 lookup tables (LUT) and 8 storage elements. Lookup tables in these series of products have 6 inputs and 2 outputs, used to program desired circuit behaviour. The chip also has a discrete CPU which is



Figure 3.6: Xilinx ZC706 development board. *Image sourced kamami.com*

a dual-core ARM Cortex A9 processor that operates at 800 MHz and the board has 2 Gigabytes of available memory (DDR3 SDRAM) [109]. In this particular design, the ARM processor is not involved in the computation of the SAD.

Programming FPGAs traditionally involves coding in hardware description languages (Verilog, VHDL). In 2012, Xilinx introduced Vivado HLS (High-Level Synthesis), an integrated design environment (IDE) with a system-to-IC-level tool. The purpose of this tool is to accelerate the development time of programmable logic blocks by synthesizing designs written in C, C++ or System C into the equivalent hardware description. In essence, Vivado HLS abstracts the finer implementation details of the final hardware design and allows developers to focus on the functionality of a system. When writing Vivado HLS code, the designers must consider how it ultimately maps to the underlying hardware components and, therefore, must frame it such that it is appropriately interpreted and ultimately synthesized into the desired circuit. Although the results of synthesis may not be as optimal as designs entirely written in Verilog or VHDL, it significantly accelerates FPGA block design and allows for rapid parameter changes. As development time is an important factor to consider for real-world application, this approach is adopted in the design of the SAD FPGA block.

3.4 Architectural Considerations and Implementation

The design is implemented with the goal of solely accelerating the SAD block matching operation. Illustrated in Figure 4.3, the ARM processor is only used to read and prepare images during testing, but have the FPGA completely responsible for the SAD. The processor and the FPGA communicate through the on-board DRAM with the help of direct memory access (DMA) blocks instantiated in the fabric of the FPGA. Every time

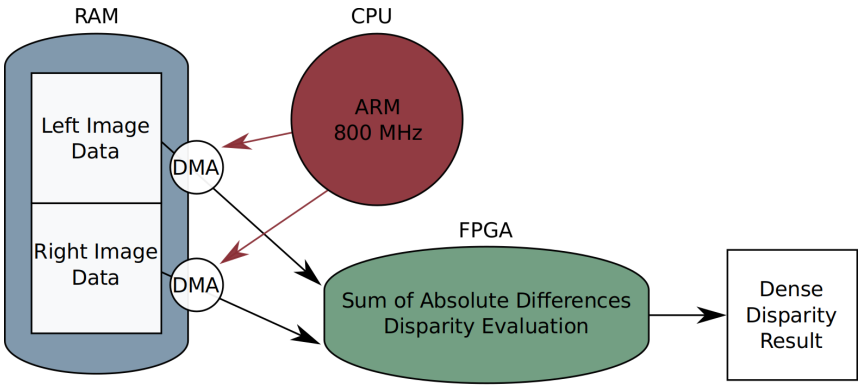


Figure 3.7: Heterogeneous system operation. The ARM CPU is only used to program and trigger the direct memory access (DMA) blocks to feed data from RAM to the SAD FPGA block. The entire disparity computation process is thus fully offloaded to the FPGA.

image data is streamed to or back from the FPGA, the CPU programs the relevant DMAs to perform the image data transfer task.

An important characteristic of the SAD FPGA block is the nature of the parallelism and the pixel memory access. Several FPGA implementations of stereoscopic vision [110–112] employ various levels of pipelining and parallelization, most of which focus on computing SAD over multiple pixels, as per Equation 3.5.

In the presented system, the SAD block operates on pixel data in-stream with the parallelism occurring at individual pixels' disparity computations. The stream of pixels and derived data passes through different computing stages without accessing external memory in between. This in-stream approach was chosen as it best compliments the raster scan readout of most modern digital image sensors and can therefore be streamlined as part of an overall pre-processing image pipeline. Additionally, its lack of access to external memory minimizes latency associated with external read and write operations. Data pipelining becomes more difficult in cases in which the number of pixel accesses may vary [113], such as when dynamically varying the window size or the disparity range.

3.4.1 Memory Management and SAD evaluations

As the SAD operation operates over a window region of pixels, it is necessary to have the relevant pixel information already available in the FPGA for the computation. Since the incoming data is in a raster stream, pixel values are buffered in the fabric of the FPGA so that access to them is still possible on subsequent clock cycles. Shown in blue in Figure 3.8, this general pixel buffering is done with 18Kb BRAMs used as line buffers. Given a SAD with window, W, of size $N \times N$, $N-1$ line buffers are used with each one buffering an entire row of the required pixel data. Additionally, for optimal resource utilization, the top and bottom rows associated with W share the same Line Buffer.

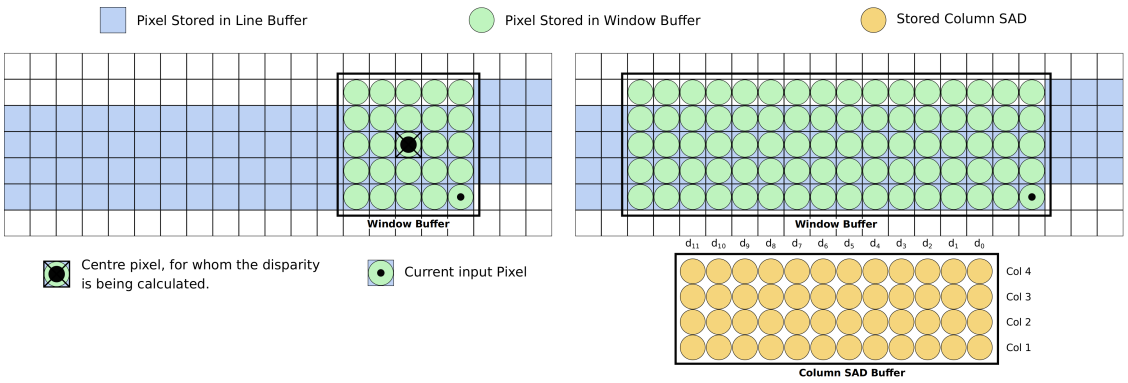


Figure 3.8: Memory elements required to efficiently evaluate 5×5 SADs over the disparity range $d_{max} = 11$. Left: Reference patch from the left image. Right: Matching region from the right image.

In Figure 3.8, the left hand portion of the figure refers the data management happening on the data from the left (reference) image whereas the right hand portion refers to the data management happening on the right (search) image. As each pixel in the left image is matched against a range of candidates in the right image, a large portion of pixel values must be available to allow for the appropriate SAD evaluations to be computed.

The window size of the depicted SAD used in the example depicted in Figure 3.8 is 5×5. As a result, to store the necessary pixel data, four separate line buffers - each occupying its own 18Kb BRAM and each storing a row's worth of pixels - are used. Each column index has its own memory space allocated in each of the four line buffers with each of these entries corresponding one of stored pixel values in that column of the image. As a new pixel is fed into the FPGA accelerator, the values in the line buffers need to update the corresponding entry they are storing at that pixel's column index. To do this, the bottom-most line buffer shifts the value it was storing to the line buffer above it at the same index and replaces it with the value of the incoming pixel. The process is repeated in the same way for the other three line-buffers; each time shifting the value they previously stored to the line above and replacing it with the value they received from the line buffer below (a value is "lost" once it is shifted out of the top-most line buffer). The result of this type of memory management with line buffers is that there are four complete rows of pixels that are stored locally on the FPGA at any given moment during the accelerator's operation. In terms of the underlying hardware resource, assuming 8-bit pixel resolution, a single 18Kb BRAM can hold a maximum of 2250 pixels. If the width of the images being processed exceeds 2250, then additional 18Kb BRAMs would be used to create wide enough line buffers to accommodate, *e.g.* $2\times$18Kb BRAMS per line buffer for images with horizontal resolutions between 2251 and 4500; $3\times$18Kb BRAMS per line buffer for images with horizontal resolutions between 4501 and 6750.

Although line buffers are an efficient means of storing large quantities of incoming pixels, they are only capable of a single read/write operation per clock cycle (due to read/write limitations of the underlying BRAM devices). As the SAD operates over a d_{max} disparity range, this would require a minimum of d_{max} clock cycles to access and compute the SAD over the required pixel values. This would significantly limit the throughput of the overall system.

Instead, depicted as green and orange circles in Figure 3.8, local instantaneous registers in the fabric of the FPGA are used to hold all the values that are needed in the computation of a given pixel's disparity. As shown in the figure, this would correspond to the 5×5 window surrounding the reference pixel in the left image, the values in the right image against which a SAD would be evaluated ()$d_{max} \times W$ in size), and separate storage of the column sums from previous SAD evaluations. To overcome the limitation of only being able to read one pixel per clock cycle from the line buffers - effectively only making a single column of the window available per clock cycle - the registers are updated in a rolling window fashion were each column of the sets of registers is shifted left every time a new pixel is received. In this way, once a window-column of values is read out from the line buffers, it is kept in register memory until it is no longer needed. Once a set of values is shifted out from the left-most column in a window, is is never read in again or needed. Combining these groups of local instantaneous registers, referred to as window buffers, with the BRAM line buffers like this allows for the efficient long-term storage of pixel values whilst simultaneously providing immediate access to all the necessary data for the computation being done at any given instant.

In addition, as previously mentioned and depicted in the lower right-hand part of Figure 3.8, to efficiently make use of the FPGA's logic resources and avoid redundant computations, the column sums from previous evaluations are also stored. This is because subsequent pixels in the reference patch share $N-1$ of N columns in their window W and, as each column is matched to every other column along the range d_{max}, $N-1$ of the columns will be recomputing the same column SAD. Therefore, for every new pixel, only the last, new column of pixels has to have the SAD computed across the range d_{max}. The other $N-1$ columns have the sum of the SADs in their column (across d_{max}) stored and accessible in local, instantaneous memory. This still requires that all the pixel data along the disparity range in the right image be kept, but it reduces the amount of absolute difference computations that are done.

As a result of this memory management, all pixel values required to compute all SADs over d_{max} are available on the same clock cycle. Therefore, by increasing logic resource utilization, all disparity evaluations for a given pixel can be computed simultaneously.

3.4.2 Minimum Evaluation

The SADs evaluated across the d_{max} range are held in a register array from which the minimum must subsequently be extracted. One of the previously proposed methods for accelerating this is based on a binary tree[114]. However, in their work, Ghosh et al. find the minimum over a block of SAD values sequentially. In a completely different setting, the maximum alignment of DNA sequences is found using a similar binary tree [115]. Finally, in Isakova et al. [111], the pipelined binary tree is used to find the minimum of SAD values.

A similar approach to Isakova *et al.* [111] for minimum finding is used, as illustrated in Figures 3.9 and 3.10. A sequence of N indexed SAD data is computed for a range of N disparities d. These data and their indices are fed into the binary tree input ports. After $O(\log N)$ stages and clock cycles, $O(N)$ comparisons are made and the SAD minimum with the corresponding index is found. Its index determines the disparity for that window.

3.4.3 Pipelining

As the design is synthesized with high-level synthesis, the tool is responsible for determining the schedule of the operations. To maintain high throughput, the amount of pipelining it imposes varies according to the timing constraints imposed by the designer. The faster the clock cycle desired, the more the tool requires to compensate for propagation delays that exceed its period. It does so by splitting the associated operations into smaller sub-operations computed over additional stages in the processing pipeline. As a consequence

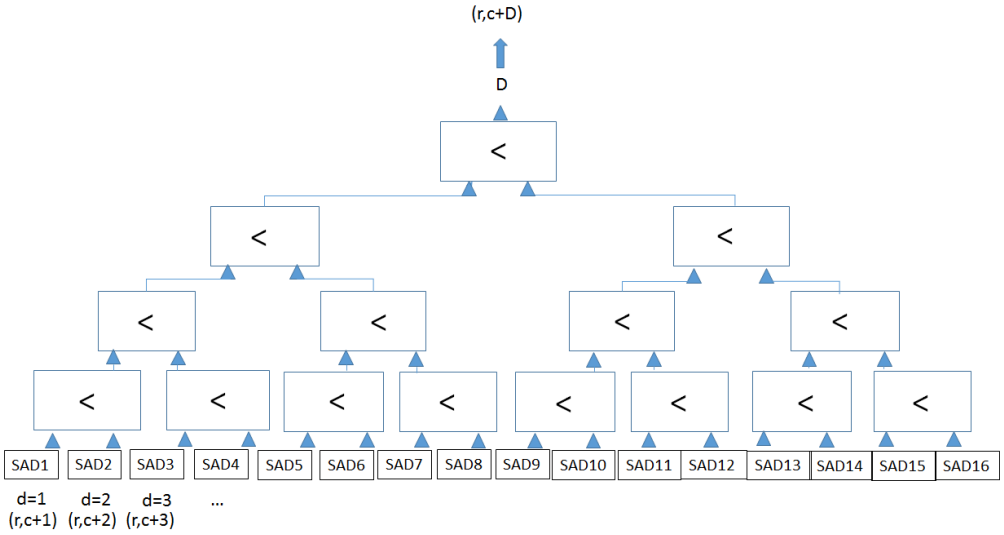


Figure 3.9: The pipelined binary tree is used to find the minimum. The block of N SAD data and their indices are fed into the binary tree input ports. After $O(\log N)$ stages, $N-1$ comparisons are made and the SAD minimum is found. Its index determines the disparity for that window.

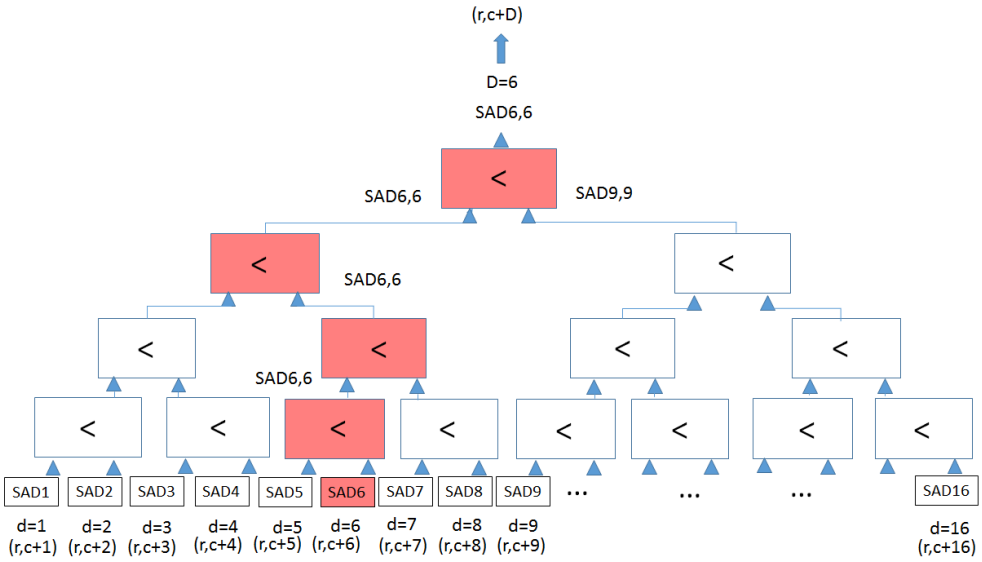


Figure 3.10: SAD minimum value path for the case of a 3×3 window, in which the minimum sum of absolute differences with respect to the pixel located at (r,c) in the left image is found 6 pixels away on the right in the right image. The pipelined binary tree is used to find the minimum SAD6. The block of N SAD data and their indices are fed into the binary tree input ports. After $\log_2 N = 4$ stages, the SAD minimum is found to be SAD6. Its index 6 means disparity $D = 6$.

of this increase in the number of pipeline stages, extra hardware resources are necessary in the form of flip flops (FF) to store intermediate values.

3.5 Results

Different variations of the SAD block were tested on the 2015 KITTI Vision Benchmark Suite [40]. The dataset contains 400 training stereo image pairs and the ground truth for 200 of them. As any implementation of the SAD accelerator is only capable of processing images of a constant resolution, only 310 out of the 400 images available are used. These 310 image pairs all have the same 1242 × 375 resolution, whereas the other 90 have different resolutions.

The accuracy of the algorithm, frame rate and resource utilization with respect to varying window sizes (keeping the disparity range constant) are first compared. The window size is varied between 5×5 and 15×15. A pixel is considered to have an incorrect disparity if its value differs by more than 3 pixels from the ground truth value and this difference is greater than 5% of the ground truth value. The error rate is then measured as the percentage of pixels with incorrect disparities over the total number of pixels.

As shown in Table 3.3, it is clear that using SADs computed over larger windows provides more accurate disparity maps. This is expected as, with larger windows, the SAD has more pixel values to match and is therefore stricter in its search. Compared to other

Table 3.1: Impact of Varying Disparity range on Frame Rate and FPGA Usage.

Disparity Range (Pixels)	75	100	125
Frame Rate (FPS)	389	386	386
Utilisation (%LUT)	18.19	22.82	27.88
Utilisation (%FF)	9.45	11.73	13.93
Power Total[FPGA] (Watts)	2.37[0.71]	2.49[0.86]	2.61[0.95]

algorithms that are listed in the benchmark, the results are far less accurate. This is also to be expected as the algorithm is purely a SAD function without any post-processing. It does not inherently enforce any smoothness across the image and is highly susceptible to incorrectly predicting disparity values in textureless or occluded regions. Additionally, the FPGA block is able to maintain a near constant frame rate despite the variation in window size. As can be observed, this comes at the expense of additional logic resource utilization.

Similar results are obtained regarding the frame rate when the maximum disparity range is varied. As shown in Table 3.1, increasing or decreasing the disparity range does not impact the output frame rate; it varies the required logic resources of the FPGA instead. The impacts are quite pronounced as increases in disparity range incur additional memory costs for both window buffer and column sum buffer.

Table 3.2: Impact of Varying Input Image Resolution on Frame Rate and FPGA usage

Image Resolution	QQVGA (160×120)	QVGA (320×240)	VGA (640×480)	KITTI (1242×375)
Frame Rate (FPS)	8955	2366	577	386
Utilisation (%LUT)	23.08	23.07	23.21	22.82
Utilisation (%FF)	11.73	11.73	11.73	11.73
Power Total[FPGA] (Watts)	2.89[1.23]	2.49[0.83]	2.49[0.83]	2.49[0.83]

A factor that does have a significant impact on the frame rate is the resolution of the stereo input pair of images. Shown in Table 3.2, as this resolution decreases, the frame rate increases. This is largely due to the streaming nature of the FPGA block. After the latency attributed to filling its internal line and window buffers, the FPGA block is capable of outputting one pixel per clock cycle. Therefore, for smaller resolutions, it is able to process many more frames given the same number of clock cycles.

Table 3.3: Impact of Varying Window Size on Frame Rate and FPGA Usage.

Window Size	5×5	7×7	9×9	11×11	13×13	15×15
Frame Rate (FPS)	397	395	388	386	388	390
Error (%)	46.60	39.01	34.39	31.38	29.35	28.01
Utilisation (%LUT)	12.28	15.37	19.56	22.82	27.42	32.35
Utilisation (%FF)	6.37	7.32	10.24	11.73	12.72	13.67
Power Total[FPGA] (Watts)	2.33[0.67]	2.27[0.60]	2.53[0.86]	2.49[0.83]	2.53[0.86]	3.07[1.40]

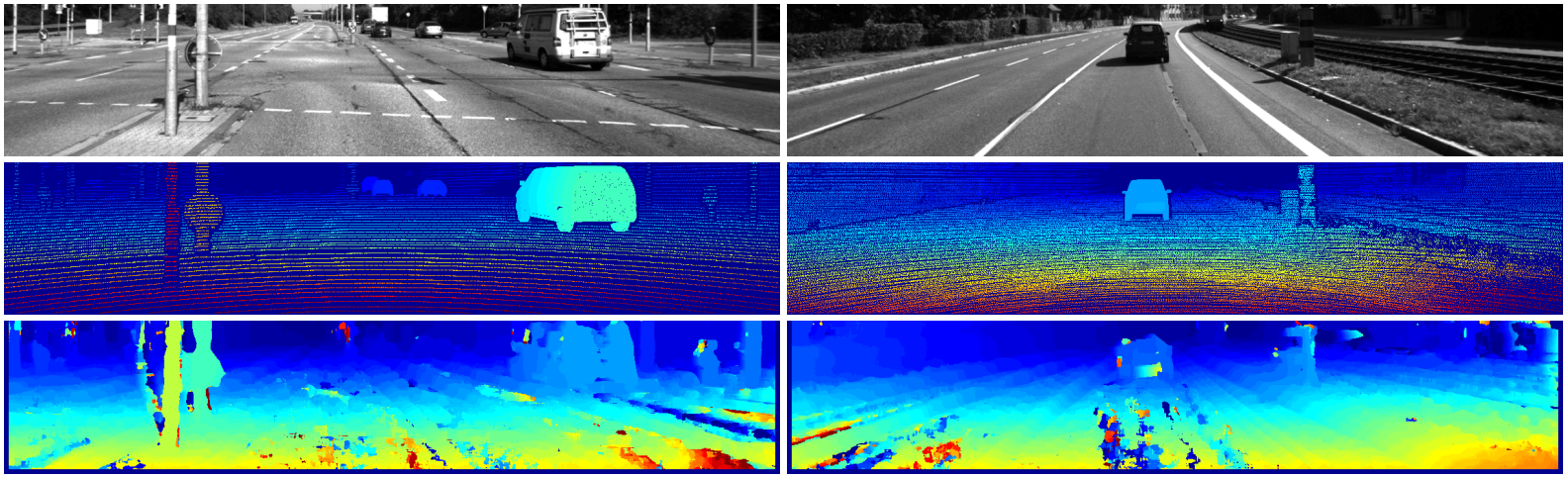


Figure 3.11: Depth Images obtained with $d_{max} = 100$ and 15×15 window size. Top to bottom: Input Image, Ground Truth, FPGA SAD Disparity Result

3.6 Discussion

As the SAD block was implemented with VHLS, varying the window size, disparity range and resolution of the stereo pair only required changes to top-level parameters. The time to recompile, resynthesize and reimplement the system is approximately 30 minutes. Thus, the design is highly flexible given the same base source code.

The only factors that impact the frame rate of the design are the resolution of the input stereo pair and the clock used on the FPGA. Ignoring the first few cycles of latency to read in the necessary minimum amount of data, the design outputs one pixel per clock cycle. Therefore the number of pixels dictates the total amount of time needed to output the corresponding result. In a similar fashion, the clock sets the speed at which these results are outputted. Increasing the clock speed also increases the number of pipelined stages in the design. This does not affect the design's throughput of one pixel per clock cycle and only increases the initial latency by the additional number of pipelined stages. For example, increasing the clock speed from 100 MHz clock to 167 MHz varies the number of pipeline stages from 2 to 17 - adding 15 additional clock cycles of initial set-up latency. This is negligible when compared to the total amount time of processing a single image which is 640×480 = 307,200 cycles.

As previously described, the implementation of the SAD block was focused on throughput. After an initial set-up period in which the line buffers are being filled, the SAD matching block outputs 1 result disparity per clock cycle until it finishes processing the entire image. The frame rates that are achieved are in excess of anything else published at the time of writing for any of the tested resolutions [102, 125]. Using the Million Disparity Estimation per second (Mde/s) metric presented in the review by Tippetts *et*

Table 3.4: Comparing proposed evaluation against other similar implementations

Method		W × H (Disp)	Time (ms)	Mde/s	KITTI FPS
Proposed		1242 × 375 (125)	2.6	**22,472.2**	**386**
Ambrosch *et al.*	[116]	450 × 375 (60)	1.7	6,062.9	104
Cuadrado *et al.*	[117]	640 × 480 (60)	3.3	5,585.5	96
Jin *et al.*	[118]	640 × 480 (60)	4.3	4,239.4	73
Kalarot *et al.*	[119]	1024 × 768 (128)	33	3,050.4	52
Woodfill *et al.*	[120]	512 × 480 (52)	5	2,555.9	44
Naoulu *et al.*	[121]	640 × 480 (60)	7.7	2,396.9	41
Perri *et al.*	[122]	512 × 512 (255)	39	1,714.0	29
Zhang *et al.*	[123]	640 × 480 (60)	16	1,152.0	20
Ambrosch *et al.*	[124]	750 × 400 (60)	16	1,152.0	20

al. [102], the presented implementation with 11×11 SAD window sizes, a 125 max disparity range and stereo input images of 1275×375 pixels (last column, Table 3.1) achieves 22,472.4 Mde/s. As shown in Table 3.4, this is $3\times$ greater than anything reported across any platform by Tippets et al. in their review[102]. The success of this approach shows promise for future real-time implementations of more complicated and more accurate algorithms.

3.7 Conclusion

The proposed approach is based on an in-stream FPGA architecture, which drastically reduces the memory access and increases the data processing rate through pipelining and parallelization. It managed to achieve very high frame rates across a range of input resolutions. Due to the high abstraction approach to the design of the system, the FPGA block is very flexible to changes in image resolution, SAD window size and disparity range. For a given input resolution, changes to the window size or disparity range does not impact the output frame rate as the approach trades FPGA resource utilization for time efficiency. Many of the other more accurate and complex stereo algorithms use the SAD as an underlying operator; such an embedded high frame rate approach sets a road map for their future real-time implementations.

"There is no authority who decides what is a good idea."
— Richard Feynman

4

Redeveloping "ELAS" for Speed and Power Efficiency

Contents

4.1 Preface 44
4.2 Introduction 44
4.3 Related work 45
4.4 Preliminaries 46
4.4.1 Original ELAS algorithm 46
4.4.2 Platform 47
4.4.3 High-level synthesis 48
4.5 System Overview 48
4.6 Key Accelerator Design Traits 50
4.6.1 FPGA memory management 50
4.6.2 Pipelining 51
4.7 Platform Conscious Algorithm Changes 52
4.7.1 Feature selection 52
4.7.2 Measuring ambiguity 56
4.7.3 Filtering support points 56
4.7.4 Data Reduction 57
4.8 Experimental Results 58
4.8.1 Accuracy 59
4.8.2 Per-frame processing time 61
4.8.3 Power and resource consumption 61
4.8.4 Throughput 62
4.9 Comparison & Discussion 63
4.10 Conclusion 64

4.1 Preface

The previous chapter demonstrated how it was possible to implement a simple but very fast stereo image processing system with a relatively small amount of FPGA resource. Albeit fast, the accuracy of this method was, however, not competitive with more advanced methods. This motivates the desire to build atop of the success in speed and readapt a more accurate algorithm.

This chapter reflects the work published in "Real-Time Dense Stereo Matching With ELAS on FPGA-Accelerated Embedded Devices" [126] with myself as the primary author. Compared to the original paper, the contents of this chapter are mostly identical with a few exceptions for the purposes of completeness and clarity. Most notably, some of the system description was expanded to provide more details along with the inclusion of a couple of new figures. The introduction of the paper has also been shortened, as it would otherwise have repeated much of what was already stated in the introduction of this thesis.

4.2 Introduction

Stereo matching algorithms can be broadly split into global energy minimization methods and local matching techniques. The former are often more accurate, but the generally large/irregular memory requirements and sequential/iterative nature of their underlying algorithms make them dependent on powerful processors for speed up, and even then, their frame rate is limited. Conversely, local methods struggle with textureless and occluded regions, but the uniformity of their computations and the absence of dependencies between pixels makes them very suitable for parallel acceleration.

Benefiting in part from the greater accessibility provided by CUDA/OpenCL, such acceleration has been predominantly achieved with Graphics Processing Units (GPU). Field Programmable Gate Arrays (FPGA) are becoming increasingly competitive alternatives, especially in power limited systems, with their capacity for in-stream processing, adherence to strict timings and supremacy at sliding-window operations [84].

Their effectiveness for stereo image processing has been previously demonstrated [45, 127, 128] with the most accurate implementations usually relying on Semi-Global Matching (SGM) [129]. In its entirety, however, SGM is ill suited to acceleration due its highly recursive and memory intense nature. As a result, FPGA systems that have implemented it have usually either only partially implemented it or sacrificed latency and throughput by relying heavily on external memory.

In this chapter, the adoption and acceleration of a competing algorithm for low-power embedded systems is investigated. The algorithm, Efficient Large-Scale Stereo (ELAS) [86], is the fastest CPU algorithm w.r.t. resolution on the Middlebury dataset [130] and one of the most accurate non-global methods. ELAS is attractive since it efficiently implements a slanted plane prior while its dense depth estimation is fully decomposable over all pixels and, hence, suitable for parallel processing. Unfortunately, the intermediate step, *i.e.* estimation of coarse scene geometry through the triangulation of support points, is a very iterative, sequential and conditional process with an unpredictable memory access pattern; making it difficult to accelerate on an FPGA.

To overcome this challenge, we propose the first stereo implementation which collaboratively utilizes both components of an embedded CPU-FPGA System on Chip (SoC) for the purpose of algorithm acceleration[1]. Other published low-power systems achieve good frame rates by limiting the algorithms they implement to those that can be fully processed by the FPGA, even when closely coupled processors are available *e.g.* [3, 47]. We, instead, seek to take advantage of both available components to efficiently accelerate the more complicated/accurate ELAS algorithm and demonstrate its feasibility for low-power systems. Accomplishing this involves offloading the different stages of the processing pipeline onto the component that best suits the computations involved. We discuss and explain the rationale and suitability of our CPU/FPGA partitioning choices, describe the key traits required in the design of efficient accelerators and detail the changes made to best adapt the algorithm to the platform. Tested on both the KITTI and New Tsukuba datasets, our system outperforms the frame rate of the original by $\sim 15-30\times$ with a rate of 47fps (1242×375 images) and, in addition, improves upon its accuracy - all with under 4W of power consumption.

4.3 Related work

The pursuit of real-time stereo began in the 1980s [131]. Initial implementations of dense stereo minimized relatively simple matching costs, *e.g.* Sum of Absolute Differences (SAD) or Sum of Squared Differences (SSD), between left and right image patches evaluated along the epipolar lines. Kayaalp and Eckman [132] were one of the first to present such a system, capable of estimating disparity over a 64 disparity range in about one second for 256×256 images.

The first system capable of at least 30 fps - on 200×200 images with a 23-pixel disparity range - was on a custom platform built from off-the-shelf components by Kanade *et al.* [133, 134]. Similarly to [132], they used a Sum of Sum of Absolute

[1]Source code available at `https://github.com/torrvision/ELAS_SoC`

Difference (SSAD) but rather than summing over the different colour channels, they summed over the six different cameras of their multi-camera system.

The first FPGA implementation of dense stereo matching used 16 Xilinx 4025 FPGAs [43]. It relied on the Census Transform (CT) [135] and computed 24 disparity levels over 320×240 images at 42 fps. Over about the next decade, FPGAs were repeatedly demonstrated as suitable platforms for dense real-time stereo. However, they mostly implemented variations of the SAD, SSAD, zero-mean SAD (ZSAD) and CT with additional noise suppressing post-processing steps [117, 122, 124, 136, 137]. The accuracy of such approaches was not typically comparable with state-of-the-art models which were formulated in global energy minimization frameworks [130, 138, 139].

Notable improvements in accuracy of FPGA implementations were made by incorporating Semi-Global Matching (SGM) [129]. For instance, Gehrig *et al.* [140] used a 3×3 window ZSAD along a 64 disparity range and minimized over 8 separate directions to run at 27 fps. Banz *et al.* [45] proposed a similar solution, but only aggregated costs over 4 directions with a rank transform [135]. SGM's larger scope managed to partly bridge the accuracy gap between strictly local operators and global optimization methods whilst remaining suitable for acceleration. SGM still has disadvantages such as large memory requirements and a fronto-parallel bias. Alternative recent approaches have shown some improvements in both frame rate and accuracy [127, 128, 141–145]; be that as it may, they still lack the accuracy of SGM.

During the same period (2000-2010), graphics processing units (GPU) began to appear as alternative platforms for algorithm acceleration [146–148]. Although they offered speedup for sliding window algorithms such as local stereo, GPUs typically underperformed and consumed more power than their FPGA counterparts [84]. They were therefore a less favourable option for truly embedded and real-time systems.

Currently, the fastest CPU stereo algorithm on the Middlebury dataset [130], normalized w.r.t. resolution, is Efficient Large-Area Stereo [86]. It competes with SGM in accuracy, but its diverse computational nature has it overlooked in favour of other fully FPGA implementable algorithms. With new closely coupled CPU/FPGA System-on-Chip devices, though, it stands to benefit in terms of acceleration.

4.4 Preliminaries

4.4.1 Original ELAS algorithm

The Efficient Large-Scale Stereo Matching (ELAS) algorithm [86] relies on the assumption that not all correspondences are *equally difficult*. It first establishes a set of sparse correspondences whose estimation is simpler and at the same time comes with

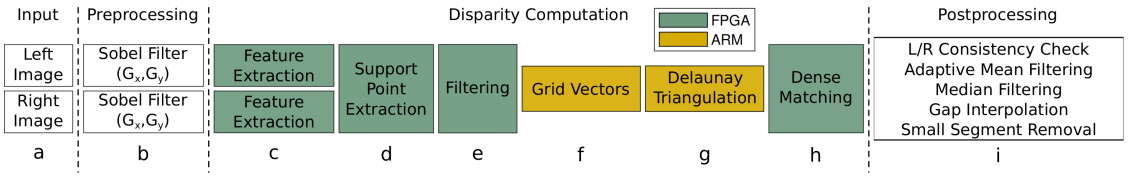


Figure 4.1: ELAS Overview: Extract a set of support points from gradient images that are then used to establish priors for the dense matching stage.

a higher degree of confidence. These correspondences provide a coarse approximation of the scene geometry and are used to define a slanted plane prior which guides the dense matching stage.

An overview of the ELAS is shown in Fig. 4.1. To obtain the set of sparse but confident correspondences (the "support points"), the stereo pair first passes through a SAD matching stage over the horizontal and vertical gradients of the images, Fig. 4.1(b-d). The resulting set is sparse as only pixels with sufficiently unambiguous disparity values are kept. This criterion is measured by comparing the distance between the first and second minima of the SAD evaluations across the disparity range. The results from this stage then undergo a further filtering procedure, Fig. 4.1(e), to remove implausible and redundant values which would respectively corrupt or unnecessarily complicate the coarse 3-D representation. This filtering process compares the sparse values to neighbours within a window region to ensure that they are consistent and removes identical values along the same row or column.

The set of support points is then used to guide the dense stereo matching stage (Fig. 4.1(h)) in two separate ways. First, the set of support points is used to define a slanted plane prior which guides the dense matching stage. To this end, ELAS uses Delaunay triangulation to construct a mesh which approximates coarse scene geometry (Fig. 4.1(g)). Second, this slanted plane prior is used to limit the disparity values evaluated during the dense matching stage. This range is expanded to include immediately neighbouring values (± 1), which gives the algorithm a chance to recover in case the initial mesh is incorrect (Fig. 4.2 and Fig. 4.1(f)).

Following the dense matching stage, ELAS uses post-processing (Fig. 4.1(i)) to invalidate occluded pixels and further improve smoothness across the image. Although post-processing plays an important role in obtaining an accurate final result, it is not a core part of the algorithm or unique to it, and therefore ignored.

4.4.2 Platform

We use Xilinx's ZC706 development board with the XC7Z045 SoC (Figure 3.6); this is a heterogeneous chip that incorporates an ARM Cortex A9 processor operating at

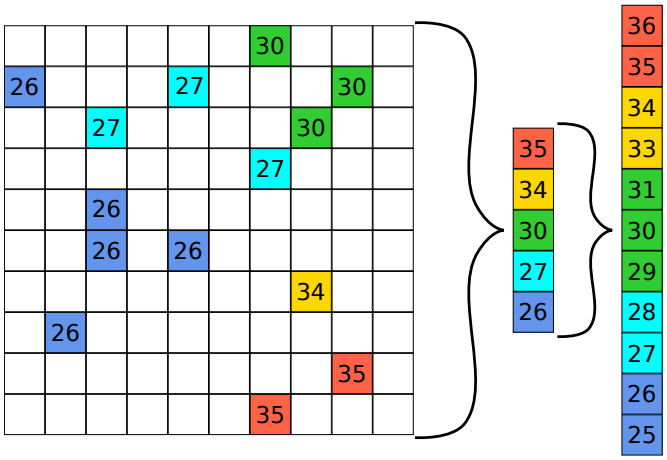


Figure 4.2: Pooling support points within a sub-region to create a grid vector. The sparse set of correspondences in every given grid region of an image are pooled together to create a characteristic search vector. This search vector is expanded to include immediate neighbours of included support points (± 1)

800 MHz and a 28 nm Kintex series FPGA on the same die. The collocation of the two components ultimately serves to increase the overall throughput of the system as it allows for rapid and efficient exchange of data. The resources available on the FPGA include: 218,600 Look Up Tables (LUT), 437,200 Flip Flops (FF), 900 DSP48 Blocks, 1090 18K Block RAMs (BRAM).

4.4.3 High-level synthesis

Vivado High-Level Synthesis (VHLS) provides a higher abstraction approach to FPGA block implementation by synthesizing designs described in a high-level language such a C/C++ into equivalent hardware descriptions. With this method, it is necessary that the designer write the high-level code in manner that is mindful of the physical hardware that it is targeting, but it alleviates the burden of adopting a low-level hardware description languages such as VHDL/Verilog. By abstracting fine-grained, less critical details of the design, VHLS accelerates and facilitates development with FPGAs. Although accelerators designed with this higher abstraction approach may not be as optimized or as resource efficient as those designed with low-level hardware description languages, the ability to deploy, modify and test them much more rapidly is a reasonable compromise. Hence, we implemented all accelerators using VHLS.

4.5 System Overview

Fig. 4.3 shows the overall pipeline illustrated in Fig. 4.1 implemented on the Zynq SoC platform. As shown, the pipeline begins by sending data (1) to an initial set of FPGA

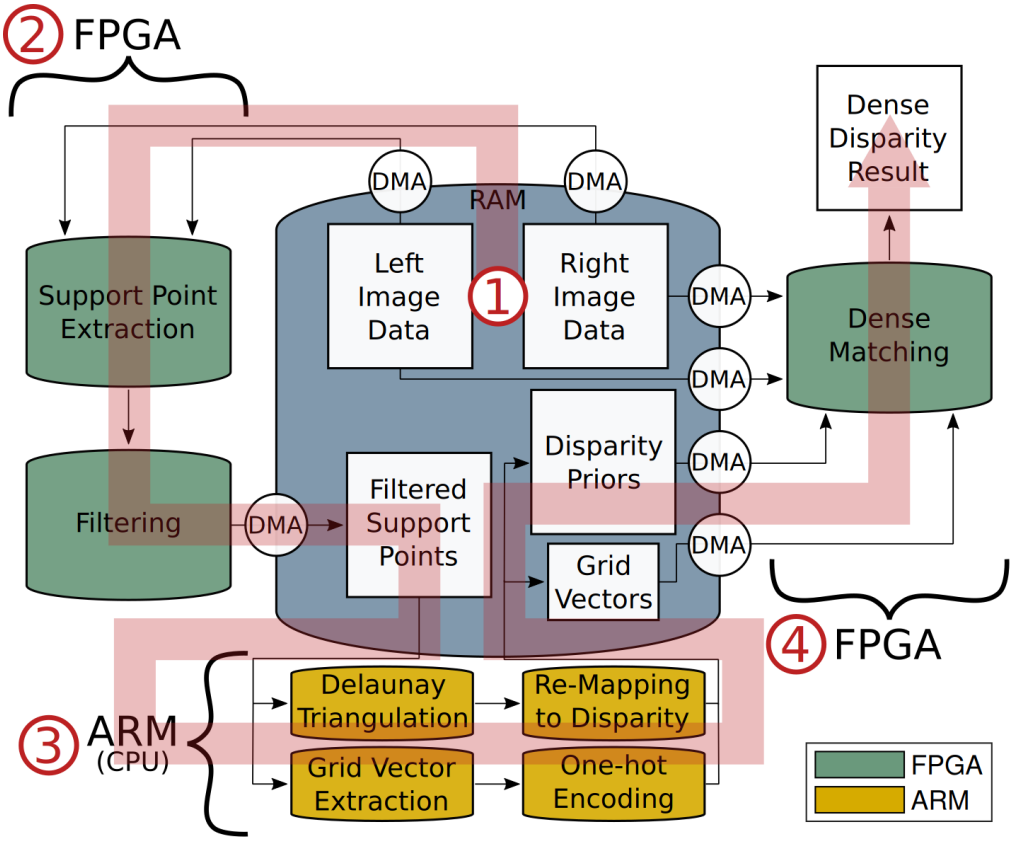


Figure 4.3: Overview of System-on-Chip (SoC) implementation. Compute-intensive tasks are offloaded to FPGA accelerators whereas conditional/sequential tasks are handled by ARM CPU. Communication between CPU and FPGA is handled by Direct Memory Access blocks in the FPGA.

accelerators (②); the results are then returned to be processed on the CPU (③); and, finally, the data is then sent for final processing to another set of FPGA accelerators (④). Determining which parts of the algorithm are offloaded onto dedicated accelerators and which are processed on the ARM CPU is a twofold process. First, the entire algorithm's CPU runtime is profiled to get an estimate of the time spent in each function. The main bottlenecks are then identified and the algorithm is mapped and broken down into its main components. In the subsequent step, the computational nature of the different components are evaluated and matched to the most appropriate component for processing.

FPGA FPGA accelerators are severely hampered if they require communication with external memory or if they contain many divergent datapaths through them. They excel at performing a variety of operations, simultaneously, on a large range of data. As such, functions that process blocks of data with well defined, relatively local, memory access patterns and limited amounts of conditional branching can benefit tremendously from such acceleration.

In ELAS, the functions responsible for support point extraction, filtering and block matching fit such criteria. Therefore, as denoted by the green blocks in Fig. 4.3 (② & ④), these are offloaded onto dedicated FPGA accelerators. These accelerators can either have data transferred in between them directly (Sparse → Filtering (①)) or to and from the RAM through Direct Memory Access (DMA) blocks in the FPGA (①) (programmed by the CPU). The accelerators process the data *in-stream* - only storing small portions of the overall stream (*cf.*, Sec. 4.6) and outputting data at the same rate at which it is received. This processing style best compliments the raster pixel readout of modern image sensors and allows for top level pipelining in between successive FPGA accelerators.

ARM CPU Functions with very unpredictable memory access patterns as well as those with a high amount of conditional branching are very ill suited for FPGA acceleration. These, instead, benefit more from the ARM CPU's faster processing speeds, its sequential processing style (invariant to branching) and its equal, but longer, access to memory (disregarding cache hit/misses).

As denoted by the yellow blocks in Fig. 4.3 (③), the ELAS processes that are handled in this manner are the Delaunay triangulation as well as the remapping of the slanted plane priors into disparity priors. Grid vector extraction and one-hot encoding (*cf.*, Sec. 4.7) are also done on the ARM. Although grid vector extraction would appear to be a candidate for FPGA acceleration as it "pools" values within a local memory region with a window like operation, as shown in Fig. 4.2, it benefits more from the faster clock of ARM since it only operates on a single value at a time.

4.6 Key Accelerator Design Traits

4.6.1 FPGA memory management

To achieve the required parallelism within the FPGA blocks, all the necessary data for a set of computations must be available on the same clock cycle as those computations are to occur. Consequently, proper memory management will have the largest impact on accelerator throughput and latency. The description and explanation of a similar type of memory management was previously detailed in Section 3.4.1 of this thesis, please refer back to that chapter for additional information. Fig. 4.4 shows the combination of block RAMs and local memory that are used as the core memory components aboard the FPGA. Block RAMs (BRAMs) are the most resource-efficient stores of large quantities of data. In the accelerators, they behave as line buffers, storing previously received pixel information into the FPGA fabric; depicted as the blue rows in Fig. 4.4. For a given $W \times W$ matching window, each image's pixel data is collected by a set of $W - 1$ line

buffers. On every clock cycle, each line buffer shifts its contents at a given index to the line buffer above it and a new pixel is read into the free space created at that index of the bottommost buffer. Similarly, the data that was in the topmost buffer is shifted out as it is no longer required - only a fraction of the overall image data is ever held in the FPGA.

Each BRAM is only able to read and write one value per clock cycle. As a result, a set of line buffers only supplies one column of pixel data per clock cycle. Most computations in the accelerators, however, operate over *windows* of pixels and therefore this alone is insufficient in meeting the memory requirements for a high throughput/low latency design. Instead, an additional $W \times W$ size "window" buffer is necessary; depicted as the green circles in Fig. 4.4. This is resource expensive as it consists of local registers within the FPGA block which are all instantaneously accessible. On every clock cycle, the contents of the window are updated by shifting all columns once to the left. The rightmost column is read in from the values stored in the line buffers (including the latest pixel value).

By combining the use of storage elements in this manner, we efficiently achieve access to all the necessary data on the same clock cycle on which it is used. No additional clock cycles need to be spent on memory access. For a more in-depth explanation, please refer back to Section 3.4.1 of this thesis.

4.6.2 Pipelining

Although VHLS handles timing considerations and data flow control of FPGA accelerators, the throughput and latency it achieves depends on the propagation delays within the accelerators as well as the desired amount of pipelining and overall clock frequency.

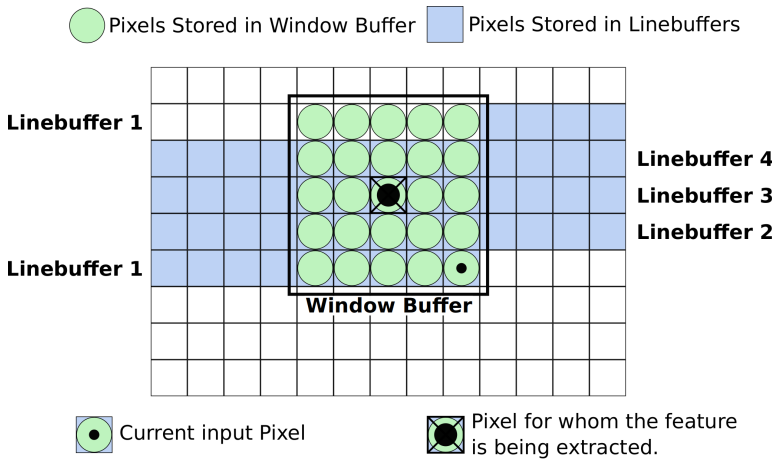


Figure 4.4: Memory requirements for sliding window operations in FPGA accelerators. Line buffers (blue) are used to store large amounts of data but can only provide a single value per clock cycle. Window buffers (green) are local registers used to store immediately required data.

When maximizing the throughput of an accelerator, pipelining is necessary when its total internal propagation delay exceeds the clock period to which it is constrained (pixel read in/out rate). By introducing pipelining, the accelerator is able to meet the clock frequency constraint by dividing and spreading its operation over multiple clock cycles. Each substage is separated from the next with flip flops that store intermediate values and therefore pipelining improves utilization. Ultimately, though, it results in the accelerator being shared across a set of inputs as each substage processes a new input on every clock cycle; this enables a larger amount of data to be acted upon simultaneously.

Other than the flip flop requirement, the trade-off with pipelining is that the number of clock cycles between the first input and output increases by the number of pipeline stages and the propagation delay experienced by a single pixel is longer as each substage's delay is extended to that of the longest substage. In image processing, though, as large quantities of pixel data pass through the accelerators, the latency introduced by pipelining is not only negligible, but significantly outweighed by the ability to output data at a much faster rate. It plays a significant role in achieving the desired frame rate in our design.

4.7 Platform Conscious Algorithm Changes

4.7.1 Feature selection

Digging into the details of the original CPU implementation of the ELAS algorithm, one of the most significant contributors to its above-average frame rate is its reliance on the SIMD capabilities of the Intel processors it was designed on. This enforces certain design choices, however, and as a result, the implementation is difficult to scale and lacks flexibility. More precisely, during both the sparse and dense feature matching stages, the original ELAS algorithm only uses specific sets of pixels from within the horizontal and vertical gradient images. These sets, depicted in Figure 4.5 and accounting for only a small fraction of the pixels in a neighbourhood, are purposely chosen in multiples of 16 as this is the maximum width of the SIMD-purposed registers (128 bits). The features used in the sparse matching stage are formed by combining four of the smaller dense features of neighbouring pixels. Although the additional information implies a higher degree of certainty and accuracy in the extraction of sparse support points, important as these form the basis for the priors subsequently computed, it also implies that these take longer to compute. To counter this, the original CPU implementation only computes candidate support points at intervals of five pixels in both the horizontal and vertical directions (a stride in x and y of 5). This not only reduces the number of available support points for the sake of speed, but also makes it difficult to scale up to higher-resolution images.

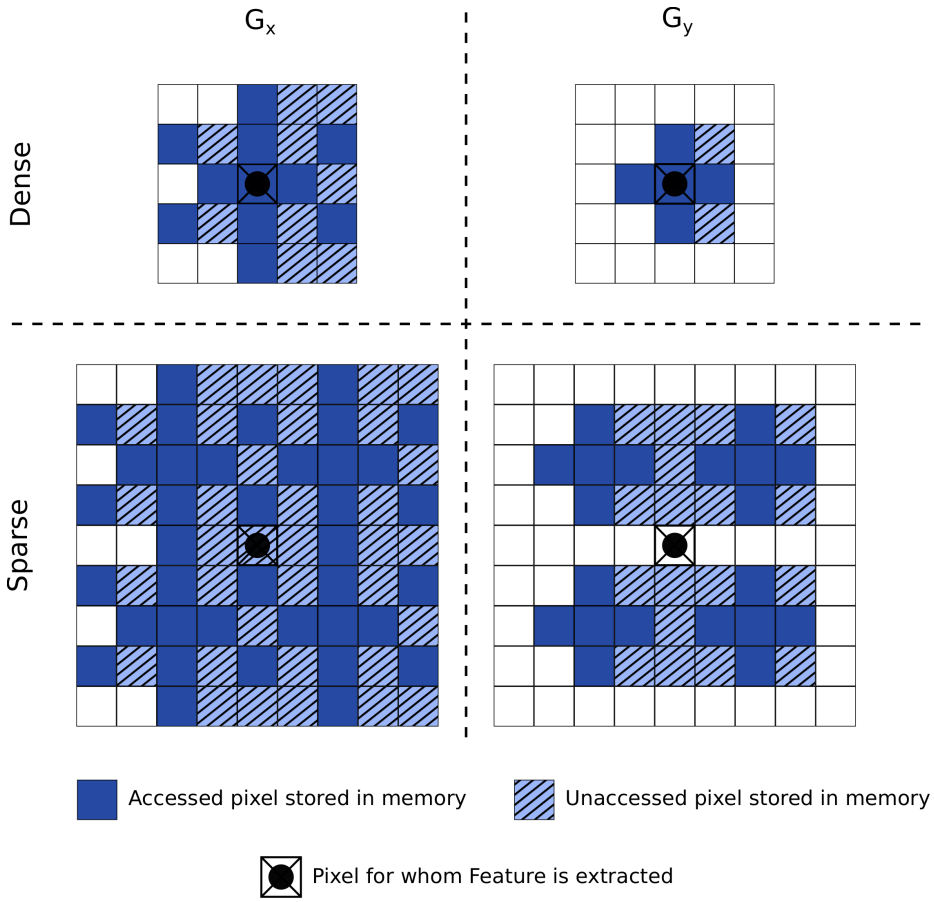


Figure 4.5: Values (dark blue) that make up both the dense features (top) and sparse features (bottom), selected from both the horizontal (G_x) and vertical (G_y) gradient images output from the pre-processing stage. Sparse features are created by combining four dense features from surrounding pixels with both features utilizing multiples of 16 values to accommodate the SIMD-purpose registers. Counter-intuitively, this design choice, intentionally made for faster CPU processing times, is inefficient and inappropriate for FPGA acceleration. In a high-throughput FPGA accelerator, most values in the relevant neighbourhood will need to be available in memory to allow for efficient pipelining. The speed of the accelerator is also independent of the number of values used.

From the perspective of accelerating the same process on an FPGA accelerator, it is equivalent to only using a subset of pixels from within a $W \times W$ window of pixels. Illustrated in Figure 4.5, accelerating such a sparse sliding-window operation would nonetheless require the vast majority of the pixels in this windows region to be available in local memory. This is due to the pipelined nature of the architecture and the one pixel per clock cycle read rate limitation of the BRAMs where these values are stored. As detailed in Section 4.6.1, this is overcome through the use of a window buffer where columns of pixels are read in from the BRAMs and shifted along in local registers until it is no longer needed. This type of memory management automatically ensures all the pixel data within the $W \times W$ window is available and therefore no additional latency is

incurred by using it (Fig 4.5). The gain in accuracy from taking advantage of a larger number of pixels during matching outweighs the additional hardware resource needed to accommodate the increased number of computations.

In addition, instead of using the same SAD based approach as the original CPU implementation, we use the Census Transform descriptors with Hamming Distance. The Census Transform (CT), originally proposed by Zabih and Woodfill [135], is a non-parametric local transform that relies on the relative ordering of local intensity values rather than on the intensity values themselves. More specifically, given a local neighborhood around a pixel $P_{(i,j)}$ at pixel coordinates (i, j), the Census Transform $R_\tau(P_{(i,j)})$ maps the surrounding values to a bit string representing the set of neighboring pixels whose intensity is less than that of $P_{(i,j)}$. With $I(P_{(i,j)})$ denoting the intensity of pixel $P_{(i,j)}$, D denoting a set of displacements for a square neighborhood around $P_{(i,j)}$ and $\otimes$ denoting concatenation, the Census Transform can be specified as,

$$R_\tau(P) = \bigotimes_{[m,n]\in D} \begin{cases} 1 & \text{if } I(P_{(i+m,j+n)}) < I(P_{(i,j)}) \\ 0 & \text{otherwise} \end{cases}$$

To compare similarity between two Census Transformed pixels, the Hamming Distance is computed. This measure counts the number of bits that differ between the two bit strings representing each pixel and its neighborhood.

Using the CT achieves illumination invariant matching without the need for an additional pre-processing step (Sobel filter Fig. 4.1(b)). Not only does this save FPGA logic resources by reducing the number of accelerators needed, but the CT inherently requires fewer resources than the SAD. Fig. 4.6 illustrates the differences in memory requirements between the two approaches; the top is the reference patch in the left image (which stays the same in both cases), the middle is the memory requirements of the search space in the righ image for the Census Transform, and the bottom is the equivalent memory requirements for the SAD instead. As shown in the middle of the figure, the CT transform for a particular pixel only has to be computed once and is then stored in a “feature” buffer from where it is reused. This "feature" buffer is a simple 1-D window buffer, *i.e.* a set of shifting instantaneous local registers, that contains values that have a bit-length equal to the Census Transform. As a new feature is calculated and shifted into this buffer, the oldest one (left-most in the figure) is being shifted out as it is not longer needed (outside of the disparity search space). Instead, with the SAD, there is continuously a need to compute new differences on the pixel data - requiring therefore for the raw pixel data to be stored and available for longer. Although the size of these pixel values generally ranges between 8-10 pixels, the vast number that need to be stored - covering the entirety of the search space in the right image - makes this a large window buffer. In addition, for

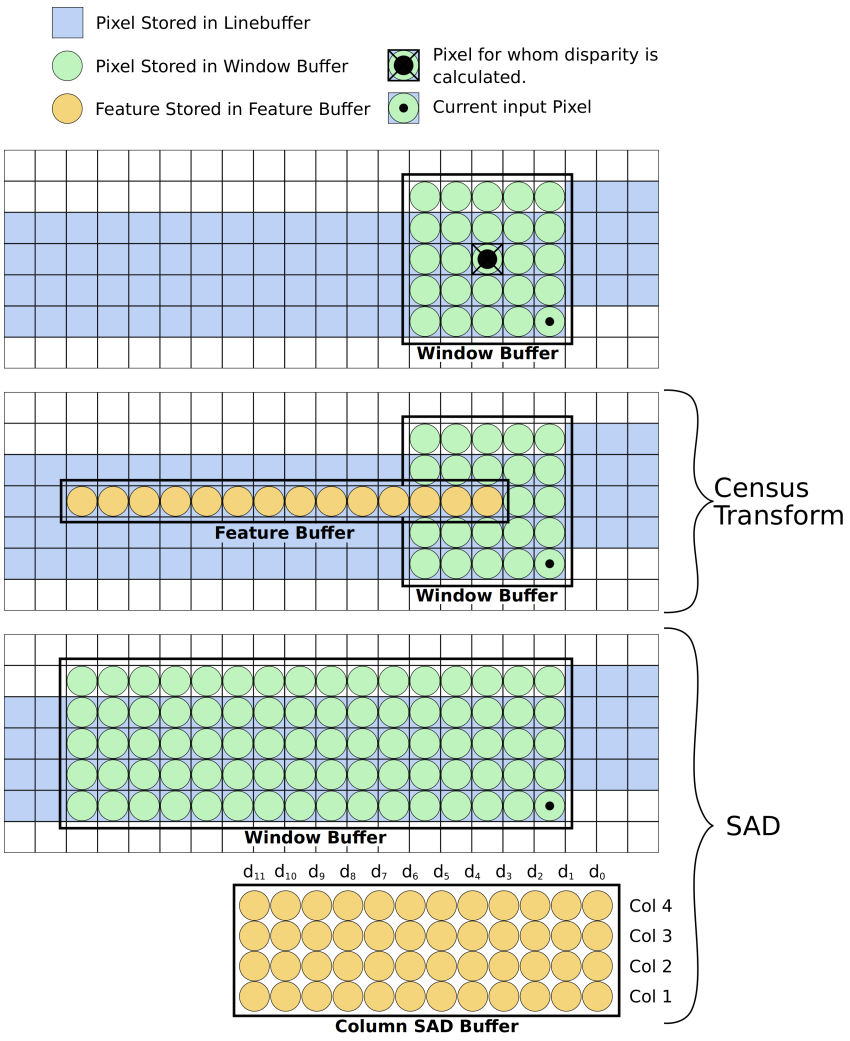


Figure 4.6: Comparison of Extraction and Matching Implementation with Census Transform and Sum of Absolute Differences

a high-throughput SAD implementation matching the speed of the CT approach, such as the one in [149] (chapter 3), there is need for another large window buffer to store previous column SAD computations (bottom of Fig. 4.6). At the same time, the number and complexity of the comparisons done as part of the SAD implementation outnumbers those that needed as part of CT implementation. For example, given the same $N \times N$ windows, the CT transform will only require to XOR N^2 bits to calculate the Hamming Distance, whereas the SAD will need to XOR at least $8N^2$ to $10N^2$ bits when computing the differences (without even accounting for logic needed for the carry-in/carry-out). Thus the resource requirement of the SAD is greater that that of the CT due to both the relatively greater number of computations as well as the greater need for memory.

As a result of these design differences made to better take advantage of the FPGA's capabilities, the new feature descriptors are both shorter in bit length whilst containing

information about larger pixel neighbourhood. More specifically, considering the same 9×9 (sparse) and 5×5 (dense) neighbourhoods used in the original implementation, the FPGA-suited features cover the entirety of both window region and are 81 bits long and 25 bits long respectively. On the other hand, the original SAD based CPU approach only considers a subset of the pixels within both window regions and the corresponding features are 512 bits long and 128 bits long respectively. The additional length due to the higher precision of the latter approach (8-bit vs. 1-bit) is less beneficial for the purposes of finding correspondences than the increased pixel coverage conferred by the former.

4.7.2 Measuring ambiguity

The support point extraction is done slightly differently to the original algorithm. We replace the original criteria which assumes a match is unambiguous if

$$\frac{m_1}{m_2} \leq 0.9, \tag{4.1}$$

where m_1 and m_2 are the first and second mimima respectively. This is equivalent to thresholding m_1

$$m_1 \leq 0.9m_2 = T_{err}(m_2), \tag{4.2}$$

Implementing such comparison on an FPGA requires a number of DSP blocks. Hence we approximate the threshold $T_{err}(m_2)$ with a shift sum

$$T_{err}(m_2) = 0.9m_2 \approx \frac{m_2}{2} + \frac{m_2}{4} + \frac{m_2}{8} + \frac{m_2}{32} = 0.90625m_2. \tag{4.3}$$

This eliminates the need for DSP blocks as shift-summing is fully accomplished within the LUT fabric of the FPGA.

4.7.3 Filtering support points

The original algorithm uses pixels both ahead and behind the one being processed during the redundancy check. As shown in the simplified 1-D example of Fig. 4.7(A), its reliance on values ahead is problematic. As these have themselves not yet been processed, they may simultaneously flag the current value as redundant and then be made redundant themselves; possibly leading to the complete extinction of a repetitive value. /mccorrectIn the example depicted in the figure, the 1-D set of data is all a repetition of the same one value. On the left, the 1-D method equivalent to the one used in the original CPU implementation would completely eliminate the value from existence. This is because it tests whether the current value is identical to the subsequent value when the redundancy

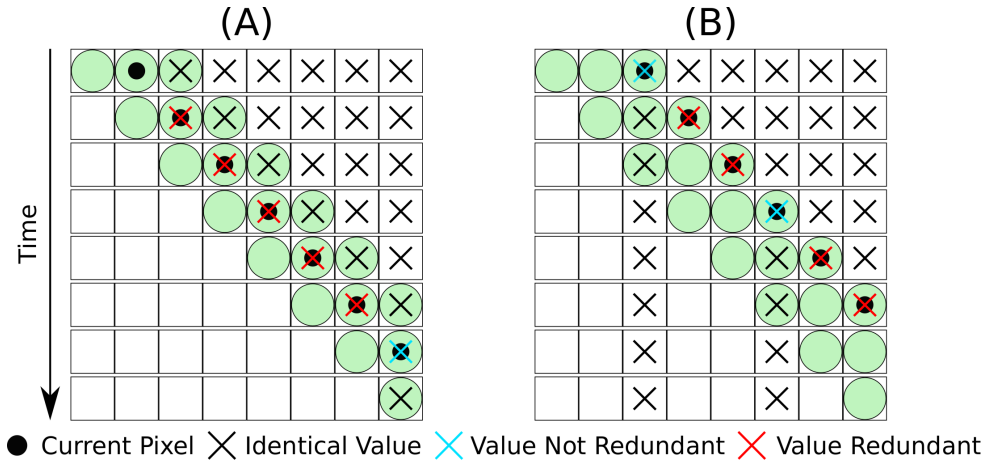


Figure 4.7: One-dimensional simplification of redundancy verification (A) Searching both forwards and backwards propagates redundancy into the value's non-existence (B) Searching strictly backwards retains the value at the desired frequency

status of the subsequent value is itself not yet known. To resolve this, the "filter" FPGA accelerator only relies on past values when determining redundancy, as shown on the right of the figure. This ensures the shared values are less frequent rather than non-existent as the process does not return to previously computed values once they have been marked redundant or not.

4.7.4 Data Reduction

To accommodate for the static bit width of accelerator ports and to minimize the data transferred to the FPGA, we introduce new data reduction steps to ELAS (on the ARM). Referring to bottom-right potion of Fig. 4.3 ③, the first one-hot encodes the grid vectors from a variable length byte array into a statically sized bitwise representation. This is done as each grid cell in the image has a different number of characteristic candidate disparities and the FPGA accelerator would otherwise need (a) additional logic to buffer this variably sized list of bytes as it is streamed in (128-bytes long in the worst case) (b) additional memory and logic to store and manage this list so that all pixels within the region retain access to it. As illustrated in Fig. 4.8, disparities within the grid vector are instead converted to high bits at the corresponding index.

This one-hot encoded format has multiple advantages. Firstly, it compresses the amount of data that needs to be sent without losing any of the relevant information. Using a fixed D_{max}-wide word to encode the list avoids the complexity of streaming a variable sized list and removes the need to store the relevant grid-vectors on the FPGA as it can be sent alongside every pixel.

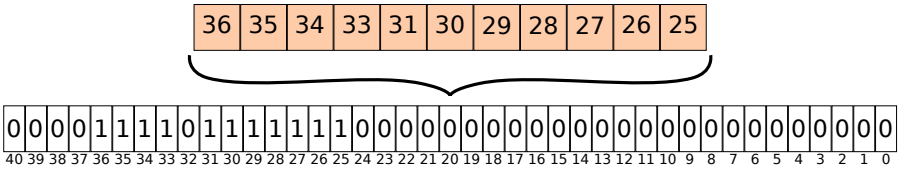


Figure 4.8: One-hot encoding of grid vectors where every disparity value is assigned a high bit at the corresponding index.

The biggest disadvantage is that the large word size may require the use of additional ports between the RAM and FPGA through which the DMAs can stream the data. The maximum width of data being streamed by an individual DMA is 64 bits. Therefore, if the desired disparity range exceeds this length, the encoded grid vector must be broken up into separate words and sent with the help of additional DMAs (as well as ports in the accelerator).

The other data reduction step, depicted in the top-right potion of Fig. 4.3 ③, converts the result of the Delaunay Triangulation from a variable mesh of triangles into a static, input image sized, matrix of disparity priors. In the original implementation, the order in which the pixels were processed was based on which triangular surface they belonged to. The memory access pattern of such a processing order is unpredictable and depends on the actual contents of the image, thus making it a poor choice to implement on an FPGA. Instead, we want the FPGA accelerator to process pixels in a in-stream manner and, therefore, the order in which the pixels should be processed is in fixed raster order; this is the reason behind the conversion. The points in this new image-sized matrix each corresponds to the disparity value of the associated pixel as interpolated from the knowledge of the triangular 3-D plane to which they belong. The pixel is streamed to the final FPGA accelerator together with its associated entry in this matrix and its above-described encoded grid-vector.

4.8 Experimental Results

To demonstrate the effectiveness of our approach, we provide a detailed evaluation across differently parameterized sets of implementations and evaluate them on both the KITTI and New Tsukuba datasets.

As FPGA accelerators cannot be easily reconfigured for different image resolutions during testing, we only use 310 of the 400 KITTI image pairs that have the same 1242×375 resolution. The New Tsukuba consists of 1800 images with resolution of 640×480, of which we use the provided subset of 200 image pairs.

4.8.1 Accuracy

We begin by verifying how the accuracy of the FPGA accelerated version of ELAS, following the modifications made to adapt it to the SoC platform, compares to that of the original algorithm. To this end, we use the standard accuracy metric from the KITTI benchmark which measures the relative number of estimated disparities which differ from ground truth by both an absolute amount of at least 3 and a relative one of at least 5%.

To ensure a fair comparison, the number of support points used to establish the prior should be approximately the same. As not all pixels are considered for support point extraction in the original algorithm, we find that this occurs when we use $\frac{1}{14}$-th of the number of total extracted support points in our method. We also use the same window sizes for both matching stages, *i.e.* 9×9 and 5×5, respectively. As previously explained, accuracy is measured without post-processing/refinement as these processes are not unique to ELAS and are more susceptible to dataset "fine-tuning". With these parameters, the original implementation tested without post-processing over the same image set, obtains an average error of 17.9% while our implementation achieves an improved 16.5%. With many other configurations tested, Table 4.1, we find that the embedded version is generally more accurate, except when window sizes are made too small or when support points are overly subsampled. Unlike the hard-coded 9×9 and 5×5 windows of the original, we are able to quickly configure different window sizes to vary accuracy without any impact on the frame rate. As larger windows provide more information in matching, they also result in more accurate depth maps. Unfortunately, this only holds up to a certain size; eventually the inherent fronto-parallel bias of square matching windows begins to negatively impact results.

Table 4.1: Impact of Window Sizes used during matching on frame rate, accuracy and resource utilization

Data Set	Window Size	Window Size	CPU ELAS		1/8		1/32		Resource Utilization [%]			Power (Watts)	
			FPS	Error %	FPS	Error %	FPS	Error %	LUT	FF	BRAM	ARM	FPGA
KITTI 1242×375	7×7	3×3	1.5 - 3	17.9	17.3	18.4	29.2	23.2	22.0	14.5	10.5	1.70	0.91
		5×5			17.4	16.5	29.2	21.7	24.2	15.3	10.8	1.70	1.06
		7×7			17.4	15.8	29.2	21.4	27.2	16.7	11.2	1.70	1.08
	9×9	3×3			12.5	17.2	24.5	20.0	26.2	17.1	10.8	1.70	1.00
		5×5			12.4	14.5	24.5	17.7	28.5	17.9	11.2	1.70	1.17
		7×7			12.3	13.7	24.3	17.0	31.7	19.3	11.6	1.70	1.16
	11×11	3×3			10.5	17.6	22.3	19.5	32.0	20.4	11.2	1.70	1.08
		5×5			10.5	14.5	22.3	16.6	34.0	21.2	11.6	1.70	1.20
		7×7			10.5	13.6	22.3	15.7	37.3	22.6	11.9	1.70	1.21
	13×13	3×3			9.4	18.2	21.0	19.4	38.5	24.2	11.6	1.70	1.12
		5×5			9.5	14.9	21.0	16.3	40.4	25.1	11.9	1.70	1.27
		7×7			9.5	13.9	21.0	15.4	43.5	26.5	12.3	1.70	1.28
Tsukuba 640×480	9×9	5×5	N/A	6.4	17.6	6.8	36.2	6.4	28.1	17.0	9.7	1.70	1.00
	11×11	7×7			14.8	6.4	32.9	5.9	37.4	21.7	10.5	1.70	1.23

Table 4.2: Time Breakdowns

Cones (900×750)	Time (ms) i7-only (orig.)	Time (ms) ARM+FPGA	Time (ms) i7+FPGA
Support Points	118	3.5	3.5
Triangulation	7	84.42	7
Matching	359	3.5	3.5

4.8.2 Per-frame processing time

Fig. 4.9 (left axes) illustrates the proportion of the overall processing time of the ARM against that of the FPGA. The FPGA portion is inclusive of the time spent transferring data by the DMAs. Additionally, the results are reported across different support point densities which we regulate through down sampling. As shown, although the time spent on the ARM is proportional to the number of support points used and shortens significantly with down sampling, it nonetheless dominates the overall processing time.

In contrast, the combined processing time of the FPGA accelerators is mostly unaffected by changes in parameters such as matching window size, disparity range or the number of support points. As they have a constant throughput of 1 pixel/clock cycle, their processing time is, instead, predominantly a function image resolution. On average, it takes only $4.84 \pm .02$ms for the KITTI images (1242×375px) and $3.19 \pm .02$ms on the New Tsukuba (640×480px). The $1.51\times$ difference corresponds exactly to the pixel ratio difference.

The line plot in Fig. 4.9 (right axes) shows the error percentage vs. the number of support points controlled by down sampling. Whilst slightly unintuitive, the best accuracy is not achieved with the largest number of support points (peak at $\frac{1}{8}$ of the support points). This is likely due to the noisy nature of sparse correspondence matching. Following this peak, accuracy gradually decreases with reduced number of support points as the resulting planar surfaces become coarser approximations of scene geometry.

Interestingly, although the FPGA accelerators run at constant time, the matching window size of the support point extraction stage is negatively correlated with overall frame rate. Larger windows do incur a greater initial latency to account for the additionally used line buffers, but this difference is negligible (evidenced by invariance of the frame rate to the window size of dense matching) and cannot account for this difference. Instead, the frame rate reduction is actually due to the increased number of support points resulting from the extraction using larger matching windows - this impacts the ARM's workload. Therefore, the main bottleneck is the Delaunay Triangulation which is in stark contrast to the one reported in the original CPU implementation.

4.8.3 Power and resource consumption

The high throughput capability of accelerators is limited by the number of circuit elements available within the FPGA fabric. In order to report this "resource utilization" (Table 4.1), we split it into the three main types of blocks (we exclude DSP blocks as they are negligibly used). As expected, the LUTs, used for the combinational logic and instantaneous memory, are the most predominantly utilized and this amount depends

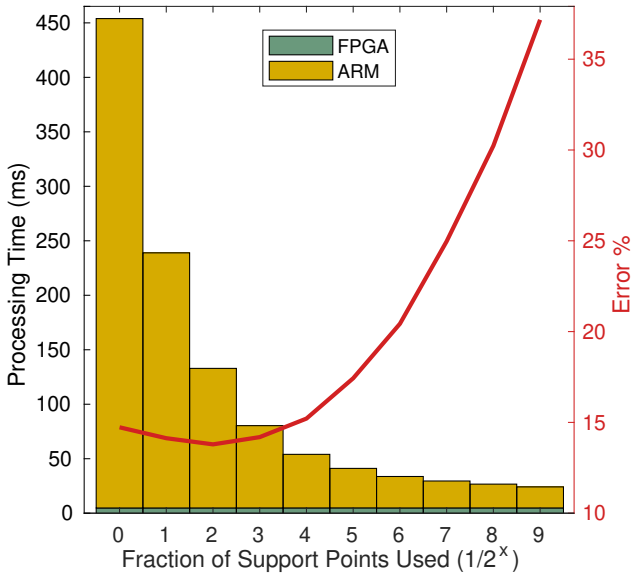


Figure 4.9: Processing time and related accuracy w.r.t down sampling

strongly on the window size of dense matching. The flip flops, utilized primarily for pipelining, share a similar dependency, but to a lesser extent. The BRAM line buffers are purely a function of the cumulative image rows required for a given set of windows.

One of the most important advantages of the proposed implementation is the power efficient computing that it enables (*cf.*, last two columns of Table 4.1). The ARM processor, running at a steady 800 Mhz, accounts for a constant but majority share of the power. In contrast, the power consumed by the FPGA is much more controlled and directly proportional to the portion of FPGA logic that is being utilized. Altogether, the results highlight that the implementation is not only capable of running the algorithm in real time, but that it also succeeds in doing this with under 3W of power (in contrast to powerful desktop CPUs which typically require $>$ 100W).

4.8.4 Throughput

To further increase frame rate, we explore operating over multiple images simultaneously by taking advantage of the additional core of the ARM and separate but identical accelerators in the FPGA. This effectively doubles the resources and power used by the FPGA (minus some shared overhead). Conversely, the ARM's power consumption remains the same, at 1.7W. For example, a dual system with (9×9) and (5×5) matching windows utilizes 56.8% of LUTs, 35% of FFs, 22.8% of BRAMs and consumes 3.67W. The most accurate configuration using this multithreaded/multi-accelerator approach runs at 23.7fps (3.74W). A faster version, whose accuracy is poorer but still better than the original CPU versions runs at 47.0fps (3.67W).

4.9 Comparison & Discussion

Comparing these results to what was achieved in the original paper, it is clear that parallelizing key parts of the algorithm has successfully led to significantly faster - up to 30× - real-time frame rates. Despite the achievement, the results also reveal some of ELAS's weaknesses for power-limited platforms. From Table 4.2, where the time breakdown for each stage is compared across systems, we see how the SoC manages 100× throughput increase for both matching stages even though it processes more data, *i.e.* all pixels considered for support point extraction and with full matching windows. However, with the low-power CPU paling in performance compared to its desktop counterpart, the triangulation procedure - seemingly insignificant in the original paper - is > 12 times longer and dominates the processing time on the SoC. Thus, although ELAS is exemplified as one of the fastest stereo algorithms, its dependence on a very sequential procedure makes it also dependent on a powerful processor to achieve maximal speed up. In the last column of Table 4.2, we show the processing times which one could obtain if a SoC, combining the same FPGA with an Intel Core i7 CPU instead of the ARM, were used to accelerate the algorithm with the same proposed approach. On such a platform, the frame rate of ELAS exceeds 70fps, but power consumption would also exceed 100W.

In terms of accuracy, the improvements over the original can be attributed to the full matching windows/CT as opposed to the randomly subsampled SAD of the original. These subsampled windows were needed in the original to speed up the CPU processing time. In contrast, with the FPGA accelerators, not only is window matching speed independent of the number of pixels considered, but full windows result in a more efficient use of resources. As a corollary, unlike CPU ELAS whose runtime is coupled to its tailored windows, our embedded version can accommodate various window sizes and disparity ranges without being concerned about the impact on latency.

We compare the performance of our system to the fastest implementation currently reported on the KITTI benchmark, referred to as "CSCT+SGM+MF" (CSM) [150] and which, at its core, implements SGM - the competing algorithm in embedded, real-time systems. It reports an 8.24% error rate at 156 fps on a 250 W NVIDIA Titan X. As CSM's result incorporates smoothing/refinement both inherently through SGM and through an additional median filter, we pass the results of our most accurate configuration through a median filter for the sake of comparison. With only this one additional post-processing step, we obtain a new error rate of 9.52% - already slightly better than the accuracy ELAS reports on the KITTI benchmark following all post-processing/refinement. Although CSM may still be marginally more accurate with a faster frame rate, it requires substantially more power at 250 W. This is equivalent to a per Watt frame rate of 0.62fps/W. In their

original paper, the authors also attempt a more power efficient implementation (same accuracy) on a mobile NVIDIA Tegra X1 GPU (10 W). After scaling their results to the KITTI resolution, this more power efficient system manages only 13.8 fps with a resulting efficiency of 1.38fps/W. Our system, in contrast, offers a 23.7fps frame rate with a corresponding 6.34fps/W efficiency. This is a $10\times$ improvement over the faster Titan X system and $5\times$ over the slower Tegra X1 one.

The FPGA accelerators in this system were described in C++ and then converted into logic with VHLS. In our experience, although the tool did eliminate the need for writing low-level VHDL/Verilog, it still relied heavily on the user's understanding of the target circuit. The original algorithm had to be completely re-engineered to comply with the hardware framework. As well as motivating the previously described modifications, this included adhering to a regular, timing-strict processing chain, minimizing any inter-process dependencies and eliminating conditional operations/branches/exceptions.

4.10 Conclusion

In this work, we have disassembled and reconstructed the ELAS algorithm onto an ARM + FPGA SoC with the purpose of evaluating its suitability for low-power, real-time embedded systems. By taking advantage of the immense parallelism available with FPGAs and by better adapting the algorithm for it, we not only successfully accelerate the frame rate by up to $30\times$, but we also demonstrate an improvement in accuracy. All this is achieved with under 4 W of power which makes it $5\times - 10\times$ more efficient, on a frame rate per Watt basis, than competing algorithms on KITTI.

This is the first time that a FPGA-CPU SoC has been used to purposely take advantage of the strengths of both devices in accelerating a stereo image processing algorithm. Deciding which portions of ELAS were accelerated on the FPGA and which were accelerated on the CPU relied on an in-depth analysis of the nature of the different sub-processes. Those that were predictable, highly parallel and with only local requirements on memory were ideal candidates for FPGA acceleration whereas those with a lot of branching, unpredictable memory accesses and strictly sequential operation were better suited to CPU processing.

Through the iterative process of adapting the algorithm to the platform as well as the starkly different processing time breakdown we obtained in the results, fundamental principles were gleaned for the future design of accurate, real-world applicable algorithms. Specifically, with parallelism being of paramount importance, any strictly sequential or iterative processes must be kept to a minimum as these will cause severe bottlenecks. Their acceleration depends on faster processors, and as CPU frequency is directly proportional

to power consumption, this quickly incurs greater power requirements that are unrealistic in space, aerial or mobile robotics. Conversely, accelerators excel at simultaneously processing vast amounts of data as long as it is available and effectively managed in the fabric of the FPGA. Therefore, compromises that may have made sense for a strictly CPU system, such as sacrificing accuracy for speed by computing with fewer pixels, are no longer necessary and should be entirely avoided.

Finally, although the newly developed high-level design tools by Xilinx do indeed facilitate the access, speed and transportability of designing on FPGAs, we found that one must still possess a strong understanding of hardware design in order to efficiently implement accelerators with them. If the code is not written with these considerations in mind from the beginning, the high-level design tools will not be successful in synthesizing a valid circuit.

Have no fear of perfection - you'll never reach it.

— Salvador Dali

5 R^{3}SGM

Contents

5.1 Preface 67
5.2 Introduction 67
5.3 Background 69
5.3.1 Semi-Global Matching (SGM) 69
5.3.2 More Global Matching (MGM) 71
5.4 Our Approach 72
5.4.1 FPGA Implementation 73
5.5 Experiments 76
5.6 Conclusion 81
5.7 Expansion to the Original Paper 83
5.7.1 Speed Bottleneck 83
5.7.2 Removing the Left Direction 84
5.7.3 Differences in Results 85
5.7.4 Summary 86

5.1 Preface

Having previously deconstructed the ELAS algorithm and re-adapted it for acceleration on a heterogeneous CPU-FPGA platform, it was clear that, although the portions of the algorithm processed by the FPGA were significantly accelerated, the more dynamic and less parallelizable portions computed on the mobile ARM CPU were significant sources of bottleneck. Not only were these portions significantly slower than the rest, but they also played an important role with regards to the accuracy of the approach. Attempting to accelerate these portions by decreasing the amount of data fed to them quickly deteriorated the quality of the output images. The motivation for the work in this chapter stems from the desire to improve the accuracy of the disparity values output by the FPGA whilst retaining the same speedup. The work builds on a well-established approach to stereo to propose a new, memory efficient and fast alternative that is ideally suited to FPGA platforms.

This chapter reflects the work published in "R^3SGM: Real-time raster-respecting semi-global matching for power-constrained systems" [151] with myself as the primary author. Although the contents of this chapter are largely identical to the originally published conference paper, a few changes have been made to include further work, help with clarity and improve structure. Most notably, the paper has been expanded with an additional section in which the work is built upon further by addressing some of the shortcomings limiting the speed of the originally proposed approach. The introduction has also been shortened to avoid repeating ideas already covered in the main introduction of the thesis. Lastly, mentions to other works that have been included in this thesis have been altered so that they point to locations in the thesis rather than to the relevant papers.

5.2 Introduction

Many binocular stereo methods estimate disparity by finding correspondences between the two images. They typically involve four phases [35]: (a) matching cost computation, (b) cost aggregation, (c) disparity optimization, and (d) disparity refinement. At a high level, such methods can be classified into two categories, based on the subset of steps mentioned above that they focus on performing effectively, and the amount of information used to estimate the disparity for each pixel:

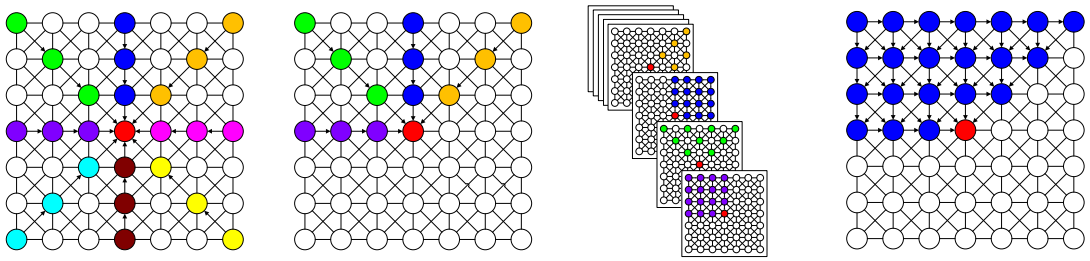
1. *Local* methods [152–154] focus on steps (a) and (b), finding correspondences between pixels in the left and right images by matching simple, window-based features across the disparity range. Whilst fast and computationally cheap, they suffer in textureless/repetitive areas, and can easily estimate incorrect disparities.

2. *Global* methods [155–158], by contrast, are better suited to estimating accurate depths in those areas, since they enforce smoothness over disparities via the (possibly approximate) minimization of an energy function defined over the whole image (they focus on steps (c) and (d)). However, this increased accuracy tends to come at a high computational cost, making these methods unsuitable for real-time applications.

Semi-global matching (SGM) [159] bridges the gap between local and global methods: by approximating the global methods' image-wide smoothness constraint with the sum of several directional minimizations over the disparity range (usually 8 or 16 directions, in a star-shaped pattern), it produces reasonable depth in a fraction of the time. It has thus proved highly popular in real-world systems, and many FPGA-based approaches have been inspired by it [45, 140, 160–163]. Other FPGA-based methods have also been presented [149, 164–167], but, whilst typically faster than those inspired by SGM, they seldom reach the same level of accuracy. Since the disparities that SGM computes for neighbouring pixels are based on star-shaped sets of input pixels that are mostly disjoint, SGM suffers from streaking in areas in which the data terms in some directions are weak, whilst those in other directions are strong. Recently, this problem has been partially addressed by an approach called More Global Matching (MGM) [168], which incorporates information from two directions into each of SGM's directional minimizations; this work was not designed with FPGAs in mind, it cannot be applied straightforwardly in an embedded context (it requires multiple passes over the pixels in the input images, several of them in non-raster order, to compute the bidirectional energies to be minimized).

In this chapter, we present an approach inspired by MGM [168] that is much more amenable to real-time FPGA implementation whilst achieving similar accuracy and much lower power consumption. We replace the multiple bidirectional minimizations of MGM, some of which cannot be computed in raster order, with a single four-directional minimization based only on pixels that are available when processing the image as a stream. This allows us to process each image in raster order and in a single pass, allowing us to stream data directly from a camera connected to the FPGA and output disparity values without requiring an intermediate buffering stage, simplifying the system architecture and reducing latency.

This paper is structured as follows. In Section 5.3, we review SGM [159] and MGM [168], the algorithms that inspired our work. In Section 5.4, we describe our method, and show how to implement it on an FPGA. Finally, in Section 5.5, we evaluate our method's accuracy on the KITTI [41, 42, 169] and Middlebury [170] datasets, and examine its power consumption and FPGA resource usage. By comparing it to other real-time stereo methods, we show that it achieves a state-of-the-art balance between accuracy, speed and power efficiency, making it desirable for use in real-time low-power systems.



(a) (b) (c) (d)

Figure 5.1: The pixels used to compute $L_{\mathbf{r}}(\mathbf{p},\cdot)$ for pixel $\mathbf{p}$ (in red) and each scan line $\mathbf{r}$, for (a) a full implementation of SGM (Section 5.3.1), and (b) a typical raster-based FPGA implementation of SGM (Section 5.3.1). Each non-red colour denotes a distinct scan line. In (c), we show the pixels that MGM would use for the same scan lines (Section 5.3.2). Finally, in (d), we show the pixels used to compute our *single* cost term, allowing us to estimate disparities whilst processing pixels in a streaming fashion (note that to compute the cost vector associated with the red pixel, we require the cost vectors only from pixels that precede it in raster order). See Section 5.4 for more details.

5.3 Background

5.3.1 Semi-Global Matching (SGM)

SGM [159] is a popular stereo matching method, owing to the good balance it achieves between accuracy and computational cost. As per [171], it aims to find a disparity map D that minimizes the following energy function, defined on an undirected graph $G = (I, \mathscr{E})$, with I the image domain and $\mathscr{E}$ the set of edges defined by the 8-connectivity rule:

$$E(D) = \sum_{\mathbf{p}\in I} C_{\mathbf{p}}(D(\mathbf{p})) + \sum_{\{\mathbf{p},\mathbf{q}\}\in\mathscr{E}} V(D(\mathbf{p}), D(\mathbf{q})) \tag{5.1}$$

Each unary term $C_{\mathbf{p}}(D(\mathbf{p}))$ denotes the 'matching cost' of assigning pixel $\mathbf{p}$ in the left image the disparity $D(\mathbf{p}) \in \mathscr{D} = [0, d_{\max}]$. This would match it with pixel $\mathbf{p} - D(\mathbf{p})\,\mathbf{i}$ in the right image, where $\mathbf{i} = [1,0]^\top$. Different matching cost functions were evaluated in [172, 173]. The choice is typically based on (i) the desired invariances to nuisances (*e.g.* changes in illumination) and (ii) computational requirements. Each pairwise term $V(D(\mathbf{p}), D(\mathbf{q}))$ encourages smoothness by penalizing disparity variations between neighbouring pixels:

$$V(d,d') = \begin{cases} 0 & \text{if } d = d' \\ P_1 & \text{if } |d - d'| = 1 \\ P_2 & \text{otherwise} \end{cases} \tag{5.2}$$

The penalty P_1 is typically smaller than P_2, to avoid over-penalizing gradual disparity changes, *e.g.* on slanted or curved surfaces. By contrast, P_2 tends to be larger, so as to more strongly discourage significant jumps in disparity.

Since the minimization problem posed by Equation 5.1 is NP-hard, SGM approximates its solution by splitting it into several independent 1D problems defined along scan lines. More specifically, it associates each pixel $\mathbf{p}$ in the image with 8 scan lines, each of which follows one of the cardinal directions (0°, 45°, …, 315°), as per Figure 5.1a. We can denote these scan lines as a vector set $R \subseteq \mathbb{R}^2$:

$$R = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\}. \tag{5.3}$$

Each pixel $\mathbf{p}$ is then associated with a directional cost $L_{\mathbf{r}}(\mathbf{p}, d)$ for each direction $\mathbf{r} \in R$ and each disparity d. These costs can be computed recursively via

$$\begin{aligned} L_{\mathbf{r}}(\mathbf{p}, d) = C_{\mathbf{p}}(d) &+ \min_{d' \in \mathscr{D}} \left(L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d') + V(d, d') \right) \\ &- \min_{d' \in \mathscr{D}} L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d'), \end{aligned} \tag{5.4}$$

in which $\mathbf{p} - \mathbf{r}$ refers to the pixel preceding $\mathbf{p}$ along the scan line denoted by $\mathbf{r}$. The minimum $L_{\mathbf{r}}$ cost associated with $\mathbf{p} - \mathbf{r}$ is subtracted from all costs computed for $\mathbf{p}$ to prevent them growing without bound as the distance from the image edge increases [159]. Having computed the directional costs, SGM then sums them to form an aggregated cost volume:

$$L(\mathbf{p}, d) = \sum_{\mathbf{r} \in R} L_{\mathbf{r}}(\mathbf{p}, d) \tag{5.5}$$

Finally, it selects each pixel's disparity using a Winner-Takes-All (WTA) approach to estimate a disparity map D^*:

$$D^*(\mathbf{p}) = \operatorname*{argmin}_{d \in \mathscr{D}} L(\mathbf{p}, d) \tag{5.6}$$

The disparities estimated by SGM only approximate the solution of the initial problem, for which we would need to enforce a smoothness term over the whole image grid, but they are much less demanding to compute and, despite causing streaking artifacts in the final disparity image, have been proven to be accurate enough for practical purposes [174].

One technique commonly used to filter out incorrect disparity values is to perform an LR consistency check [159], which involves computing the disparities not just of pixels in the left image, but also in the right image, and checking that the two match (*e.g.* that if $\mathbf{p}$ in the left image has disparity d, then so does pixel $\mathbf{p} - d\mathbf{i}$ in the right image). Observe that the disparities of pixels in the right image have the opposite sign, *i.e.* that assigning pixel $\mathbf{p}'$ in the right image a disparity of d matches it with pixel $\mathbf{p}' + d\mathbf{i}$ in the left image.

Whether LR consistency checking is used or not, SGM has drawbacks: (i) it suffers from streaking in textureless/repetitive regions, which the LR checks mitigate but do

not solve, (ii) there is a need to store the entire unary cost image (or images, when checking), to allow the computation of the directional contributions to the final cost, and (iii) there is a need for multiple passes over the data, to recursively compute the directional components used in Equation 5.5. To deploy SGM on a limited-memory platform, *e.g.* an FPGA, some compromises must be made, as we now discuss.

SGM on FPGAs

Some of the first implementations of SGM that were deployable on FPGA platforms were the ones by Gehrig *et al.* [140] and Banz *et al.* [45]. As the computation of the directional costs for a pixel requires us to have already evaluated the cost function for all pixels along the scan line (from the edge of the image), most FPGA implementations focus only on the scan lines that would be completely available when evaluating a pixel. If pixels in the images are available in raster order, then these will be the three scan lines leading into the pixel from above, and the one leading into it from its left (see Figure 5.1b). Observe from Equation 5.4 that to compute the directional costs for a pixel $\mathbf{p}$ along a scan line $\mathbf{r}$, only its unaries $C_{\mathbf{p}}$ and the cost vector $L_{\mathbf{r}}(\mathbf{p}-\mathbf{r},\cdot)$ associated with its predecessor $\mathbf{p}-\mathbf{r}$ are required.

Memory requirements are also a constraint for implementations on accelerated platforms: when processing pixels in raster order, temporary storage is required for the directional costs $L_{\mathbf{r}}$ associated with every predecessor of the pixel being evaluated, so the more scan lines we consider, the more storage is required from the FPGA fabric. Due to the limited resources available on FPGAs, algorithms such as [45, 140] choose to limit the number of scan lines that are considered. This is not only necessary to allow the processing of pixels in raster order, but also to keep the complexity of the design low enough to be deployed on the circuits. Despite attempts by a number of real-time FPGA-based approaches to mitigate this [127, 128, 141–145], this choice negatively impacts depth quality.

5.3.2 More Global Matching (MGM)

The streaking effect that afflicts SGM is caused by the star-shaped pattern used when computing the directional costs (see Figure 5.1a): this makes the disparity computed for each pixel depend only on a star-shaped region of the image. To encourage neighbouring pixels to have similar disparities, SGM relies on their 'regions of influence' overlapping; if the areas in which they overlap are uninformative (*e.g.* due to limited/repetitive texture), this effect is lost. As a result, if the contributions from some scan lines are weak, whilst those from others are strong, the disparities of pixels along the stronger lines will tend to

be similar, whilst there may be little correlation along the weaker scan lines: this can lead to streaking. This is an inherent limitation of SGM, and one that is only accentuated by removing scan lines, as is common when deploying SGM on an FPGA.

Recently, Facciolo *et al.* [168] presented an extension of SGM that reduces streaking by incorporating information from multiple directions into the costs associated with each scan line. To do this, they modify Equation 5.4 to additionally use the cost vectors of pixels on the previous scan line: see Figure 5.1c. More specifically, when computing the cost vector $L_{\mathbf{r}}(\mathbf{p},\cdot)$ for pixel $\mathbf{p}$ and direction $\mathbf{r}$, they make use of the cost vector computed for the pixel $\mathbf{p}-\mathbf{r}^{\perp}$ "above" it, where "above" is defined relative to $\mathbf{r}$, and has the usual meaning when $\mathbf{r}$ is horizontal. Equation 5.4 then becomes:

$$L_{\mathbf{r}}(\mathbf{p},d) = C_{\mathbf{p}}(d) + \frac{1}{2}\sum_{\mathbf{x}\in\left\{\mathbf{r},\mathbf{r}^{\perp}\right\}} \min_{d'\in\mathscr{D}} \left(L_{\mathbf{r}}(\mathbf{p}-\mathbf{x},d') + V(d,d')\right) \tag{5.7}$$

This approach was shown [168] to be more accurate than SGM, whilst running at a similar speed. Unfortunately, MGM's inclusion of an additional direction of optimization per cost vector renders it even more challenging to accelerate than SGM: not only are the memory and computational requirements greater, but it is also significantly more difficult to parallelize as calculating the new cost vectors is no longer separable (due to their dependency on two orthogonal directions of optimization). For example, unlike SGM which can process the vertical (90°) cost vectors for every pixel in a row simultaneously (as each column is independent in that direction), MGM's inclusion of the additional orthogonal (180°) direction means that each pixel in the row now must have access to its neighbour's result before being able to be computed. For this reason, MGM cannot easily be sped up to obtain a real-time, power-efficient algorithm.

5.4 Our Approach

MGM [168] is effective at removing streaking, but since all but two of its directional minimizations (the purple and green ones in Figure 5.1c, for which $\mathbf{r} = [1,0]^{\top}$ and $\mathbf{r} = [1,-1]^{\top}$) rely on pixels that would be unavailable when streaming the image in raster order, a *full* FPGA implementation of the algorithm is difficult to achieve (see Section 5.3.1). One solution is to implement a cut-down MGM that only uses those of its directional minimizations that do work on an FPGA (*i.e.* the purple and green ones), mirroring one way in which SGM has been adapted for FPGA deployment [45]. However, if we limit ourselves to only one of MGM's directional minimizations (*e.g.* the purple one), then the 'region of influence' of each pixel shrinks, resulting in poorer

disparities, and if we use both, then we are forced to use double the amount of memory to store the cost vectors (see Section 5.3.1).

To avoid both problems, we propose a compromise, inspired by the way in which MGM allows each directional cost for a pixel to be influenced by neighbouring pixels in more than one direction to mitigate streaking. Our approach uses only a *single* directional minimization, but one that incorporates information from *all* of the directions that are available when processing in raster order. This approach is inherently raster-friendly, and requires a minimal amount of memory on the FPGA. When processing the image in raster order, we compute the cost vector for each pixel by accumulating contributions from 4 of its 8 neighbours that have already been visited and had their costs computed (the left, top-left, top and top-right neighbours, as per Figure 5.1d). Formally, if we let

$$X = \{\rightarrow, \searrow, \downarrow, \swarrow\} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right\}, \tag{5.8}$$

then we can compute the cost vector $L(\mathbf{p}, \cdot)$ for pixel $\mathbf{p}$ via:

$$L(\mathbf{p}, d) = C_{\mathbf{p}}(d) + \frac{1}{|X|} \sum_{\mathbf{x} \in X} \left(\min_{d' \in \mathscr{D}} \left(L(\mathbf{p} - \mathbf{x}, d') + V(d, d') \right) - \min_{d' \in \mathscr{D}} \left(L(\mathbf{p} - \mathbf{x}, d') \right) \right) \tag{5.9}$$

Since, unlike SGM and MGM, we only use a single minimization, this is like Equation 5.5 in those approaches, letting us obtain each pixel's cost vector in a single pass over the image.

In our implementation, we use the Census Transform (CT) [175] to compute the unaries $C_{\mathbf{p}}$. CT is robust to illumination changes between the images, and can be computed efficiently and in a raster-friendly way (see Section 5.4.1). Moreover, the Hamming distance between two CT feature vectors can be computed efficiently, and provides a good measure of their similarity. We compute the pixel costs $L(\mathbf{p}, d)$ simultaneously for both the left and right images, thanks to our FPGA implementation (see Section 5.4.1). After selecting the best disparity in each image with a Winner-Takes-All (WTA) approach, we process the disparities with a median filter (in a raster-friendly way) to reduce noise in the output. Finally, we validate our disparities with a standard LR check to discard inconsistent results [159], using a threshold of 1 disparity or 3%, whichever is greater.

5.4.1 FPGA Implementation

Having described our approach conceptually, we now show how to implement it on an FPGA. By contrast to the previous sections, in which we only showed how to

compute the disparities for pixels in the left image, here we describe how to compute the disparities for the pixels in both images efficiently to support LR consistency checking (see Section 5.3.1). Notationally, we distinguish between the unary costs and cost vectors for the two images using the superscripts $(\mathscr{L})$ and $(\mathscr{R})$.

Two main steps are involved: (i) the computation of the unary costs $C_{\mathbf{p}}^{(\mathscr{L})}(\cdot)$ and $C_{\mathbf{p}}^{(\mathscr{R})}(\cdot)$, and (ii) the recursive computation of the cost vectors $L^{(\mathscr{L})}(\mathbf{p},\cdot)$ and $L^{(\mathscr{R})}(\mathbf{p},\cdot)$. Implementing these steps efficiently on an FPGA requires understanding the hardware, which essentially consists of a set of programmable logic blocks that can be wired together in different ways and independently programmed to perform different functions. This architecture naturally lends itself to an efficient pipeline-processing style, in which data is fed into the FPGA in a stream and different logic blocks try to operate on different pieces of data concurrently in each clock cycle. In practice, the steps we consider here all involve processing images, with the data associated with the images' pixels being streamed into the FPGA in raster order, as we now describe.

Unary Computation

Each unary $C_{\mathbf{p}}^{(\mathscr{L})}(d)$, which denotes the cost of assigning pixel $\mathbf{p}$ in the left image a disparity of d, is computed as the Hamming distance $\mathscr{H}$ between the feature vector $\phi^{(\mathscr{L})}(\mathbf{p})$ of pixel $\mathbf{p}$ in the left image and the feature vector $\phi^{(\mathscr{R})}(\mathbf{p}-d\mathbf{i})$ of pixel $\mathbf{p}-d\mathbf{i}$ in the right image. ($\phi^{(\mathscr{L})}(\mathbf{p})$ is computed by applying the Census Transform [175] to a $W \times W$ window around $\mathbf{p}$ in the left image, and analogously for $\phi^{(\mathscr{R})}$.) Conversely, $C_{\mathbf{p}}^{(\mathscr{R})}(d)$ becomes the Hamming distance between $\phi^{(\mathscr{R})}(\mathbf{p})$ and $\phi^{(\mathscr{L})}(\mathbf{p}+d\mathbf{i})$.

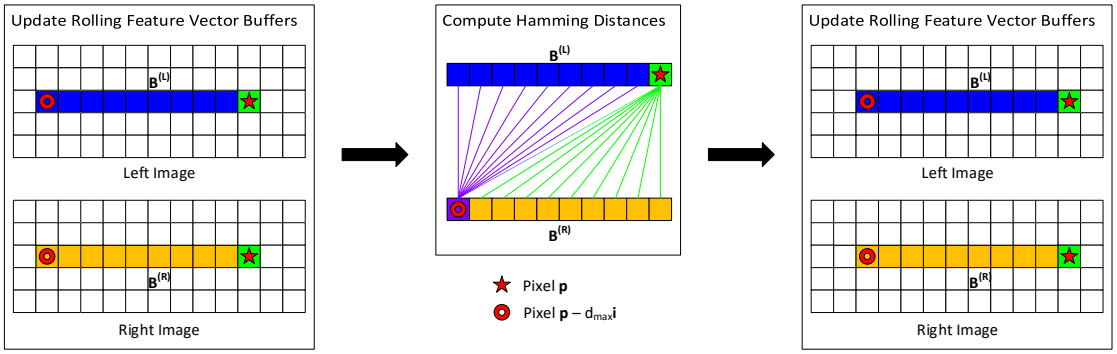


Figure 5.2: Computing the unaries: at each pixel $\mathbf{p}$, we compute $\phi^{(\mathscr{L})}(\mathbf{p})$ and $\phi^{(\mathscr{R})}(\mathbf{p})$ and use them to update the rolling buffers $B^{(\mathscr{L})}$ and $B^{(\mathscr{R})}$. We then compute the unaries $C_{\mathbf{p}}^{(\mathscr{L})}(d)$ and $C_{\mathbf{p}-d_{\max}\mathbf{i}}^{(\mathscr{R})}(d)$ for all $d \in \mathscr{D}$ as the Hamming distances between the relevant feature vectors (see Equation 5.10) before moving on to the next pixel.

As per Figure 5.2, we traverse both the left and right images simultaneously in raster order, computing $\phi^{(\mathscr{L})}(\mathbf{p})$ and $\phi^{(\mathscr{R})}(\mathbf{p})$ for each pixel $\mathbf{p}$ as we go, and maintaining rolling

buffers $B^{(\mathscr{L})}$ and $B^{(\mathscr{R})}$ of feature vectors for the most recent $d_{\max}+1$ pixels in each image, *i.e.* $B^{(\mathscr{L})} = [\phi^{(\mathscr{L})}(\mathbf{p}-d\mathbf{i}) : d \in \mathscr{D}]$, and analogously for the right image. After computing the feature vectors for pixel $\mathbf{p}$, we compute unaries for all $d \in \mathscr{D}$:

$$\begin{aligned} C_{\mathbf{p}}^{(\mathscr{L})}(d) &= \mathscr{H}(\phi^{(\mathscr{L})}(\mathbf{p}), \phi^{(\mathscr{R})}(\mathbf{p}-d\mathbf{i})) \\ C_{\mathbf{p}-d_{\max}\mathbf{i}}^{(\mathscr{R})}(d) &= \mathscr{H}(\phi^{(\mathscr{L})}(\mathbf{p}+(d-d_{\max})\mathbf{i}), \phi^{(\mathscr{R})}(\mathbf{p}-d_{\max}\mathbf{i})) \end{aligned} \tag{5.10}$$

Note that we compute the unaries for right image pixels just before they leave $B^{(\mathscr{R})}$, since it is only at that point that we have accumulated the feature vectors for all of the relevant left image pixels in $B^{(\mathscr{L})}$ (see Figure 5.2).

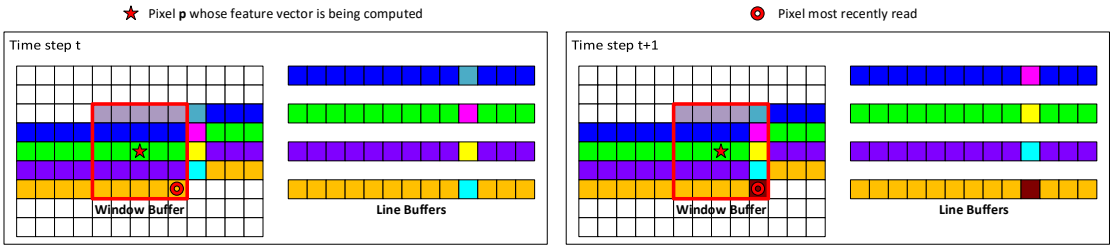


Figure 5.3: The buffers used to compute the Census Transform feature vectors, and how they are updated (see Section 5.4.1).

In practice, to efficiently compute the feature vectors, we must maintain a $W \times W$ window of the pixels surrounding $\mathbf{p}$ that can be used to compute the Census Transform [175]. As shown in Figure 5.3, we store this in a window buffer (local registers on an FPGA that can be used to store data to which we require instantaneous access). To keep the window buffer full, we must read ahead of $\mathbf{p}$ by slightly over $\lfloor W/2 \rfloor$ rows. Separately, we maintain pixels from the rows above/below $\mathbf{p}$ in line buffers (regions of memory on an FPGA that can store larger amounts of data but can only provide a single value per clock cycle). As shown in Figure 5.3, some pixels are in both the window buffer and one of the line buffers. When moving from one pixel to the next, we update the window buffer and line buffers as shown in Figure 5.3. Notice how the individually marked pixels in the line buffers are shifted upwards to make way for the new, brown pixel that is being read in (to both the orange line buffer and the window buffer), and how the turquoise pixel is removed from the blue line buffer but added to the top right of the window buffer. All of these operations can be implemented very efficiently on an FPGA.

Cost Vector Computation

Once the unaries have been computed, the next step is to compute the $L(\mathbf{p},d)$ values (*i.e.* the cost vector) for each pixel using Equation 5.9. This again involves a walk over the

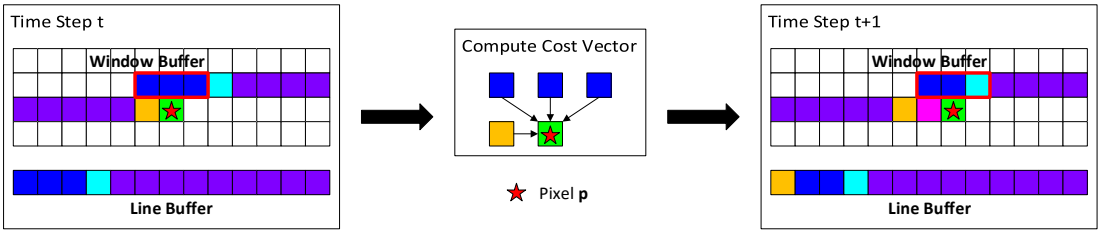


Figure 5.4: The buffers used to compute the cost vectors, and how they are updated (see Section 5.4.1).

image domain in raster order. In this case, computing the cost vector for each pixel $\mathbf{p}$ uses the cost vectors of the pixels $\mathbf{p}-\mathbf{x}$, for each $\mathbf{x} \in X$ (*i.e.* the 3 pixels above $\mathbf{p}$ and the pixel to its left). As a result, these must be in memory when the cost vector for $\mathbf{p}$ is computed.

In practice, as shown in Figure 5.4, we divide the relevant cost vectors between several different locations in memory: (i) a line buffer whose size is equal to the width of the image, (ii) a window buffer that holds the cost vectors for the 3 pixels above $\mathbf{p}$, and (iii) a register that holds the cost vector for the pixel to its left (the yellow pixel in Figure 5.4). This provides us with instantaneous access to the cost vectors that we need to compute the cost vector for $\mathbf{p}$, whilst keeping track (via the line buffer) of the cost vectors that will be required during the processing of future pixels. When moving from one pixel to the next, we update the window buffer and line buffer as shown in Figure 5.4. For the actual computation of $L(\mathbf{p},d)$, we rewrite Equation 5.9 as follows:

$$\begin{aligned} L(\mathbf{p},d) = C_{\mathbf{p}}(d) + \frac{1}{|X|}\sum_{\mathbf{x}\in X}\Bigg(& \\ \min\Big\{ & L(\mathbf{p}-\mathbf{x},d), L(\mathbf{p}-\mathbf{x},d-1)+P_1, \\ & L(\mathbf{p}-\mathbf{x},d+1)+P_1, \min_{d'\in\mathscr{D}}\left(L(\mathbf{p}-\mathbf{x},d')+P_2\right)\Big\} \\ - \min_{d'\in\mathscr{D}} & \left(L(\mathbf{p}-\mathbf{x},d')\right)\Bigg) \end{aligned} \tag{5.11}$$

This allows for a more optimal implementation in which we store $\min_{d'\in\mathscr{D}}(L(\mathbf{p}-\mathbf{x},d')$ to avoid repeat computations.

5.5 Experiments

We evaluate our approach on the KITTI [41, 169] and Middlebury [35, 139] datasets. We then compare our frame rate across different resolutions and disparity ranges vs. competing FPGA-based approaches. Finally, we break down the FPGA resource costs, associated power consumption, as well as accuracy of our approach for several variations of our design.

On KITTI, we compare our approach to the only two published approaches from the benchmark that are able to achieve state-of-the-art performance in real time [150, 176], both of which require a powerful GPU (an Nvidia GTX Titan X) to run. Since, unlike these approaches, our approach does not naturally produce disparities for every pixel in the image, we interpolate as specified by the KITTI evaluation protocol in order to make our results comparable with theirs. As shown in Table 5.1, with these approaches we are able to achieve post-interpolation results that are competitive on accuracy whilst significantly reducing the power consumption and the compute power required. Furthermore, compared to the additional power efficient implementation reported in [150] (on a Nvidia Tegra X1), which achieves 13.8 fps with 10 Watts when scaled to the KITTI dataset, our system is more than five times faster, whilst consuming less than a third of the power. Moreover, we are able to achieve an even better error rate (4.8%) pre-interpolation, with a density of 85%. For some applications, this may in practice be more useful than having poorer disparities over the whole image. Qualitative examples of our approach on KITTI frames are shown in Figure 5.5.

On Middlebury, we show in Table 5.2 and Table 5.3 that we are able to achieve comparable accuracy to a number of other FPGA-based methods, whilst either running at a much higher frame rate (c.f. [140, 165]), using simpler, cheaper hardware (c.f. [162]) or handling greater disparity ranges (c.f. [164, 166]). Figure 5.6 shows some examples of our approach on the four most commonly used Middlebury images, namely *Cones*, *Teddy*, *Tsukuba* and *Venus*.

The FPGA accelerators are designed and implemented through Xilinx's High-Level Synthesis (HLS) tool, whose C++ abstraction to FPGA development allows for faster prototyping during the design process, greater flexibility as well as improved reusability of the resulting hardware blocks. Through the use of Xilinx's SDSoC tool, we deploy the accelerators on the ZC706 development board. In Table 5.4, we highlight how the frame rate of our system is independent of the window size of the Census Transform that we employ. Varying these parameters has, instead, an effect on the quality of the estimated disparities: as the window size increases, the error rate on the KITTI dataset images [41, 169] decreases. As in Table 5.1, we report the error rate of the pixels surviving LR check (*i.e.* the output of the proposed method) together with their density, and the error rate after an interpolation step done according to the KITTI protocol. As expected, variations in the CT window size also affect the FPGA resource utilization of the system, *i.e.* the number of logic/memory units that are required to implement the necessary hardware blocks. This resource utilization, in turn, impacts the overall amount of power consumed by the FPGA chip, as estimated by the Xilinx Vivado tool. This is also shown, in the last row of Table 5.4.

Method	D1 Valid	Density	D1 Interpolated	Runtime	Environment	Power Consumption (W) (approx.)
Ours	4.8%	85.0%	9.9%	0.014s	FPGA (Xilinx ZC706)	3
DeepCostAggr [176]	–	99.98%	6.3%	0.03s	Nvidia GTX Titan X	250
CSCT+SGM+MF [150]	–	100%	8.2%	0.006s	Nvidia GTX Titan X	250

Table 5.1: The quantitative results of our approach, in comparison to state-of-the-art GPU-based real-time methods, on the Stereo 2015 subset of the KITTI dataset [41, 169]. D1 Valid: error rate on the pixels surviving the LR check; Density: % of pixels output by the algorithm (in our case after the LR check); D1 Interpolated: error rate after interpolating according to the KITTI protocol. We use a threshold of 3 disparity values or 5%, whichever is greater (*i.e.* the standard thresholds for KITTI). Our approach is able to produce post-interpolation results that are within striking distance of existing methods, whilst being two orders of magnitude more power-efficient and requiring much less computational power.

Method	Cones			Teddy			Tsukuba			Venus		
	non-occ.	all	disc.	non-occ.	all	disc.	non-occ.	all	disc.	non-occ.	all	disc.
Ours	**3.4**	**8.9**	10.3	8.2	14.6	22.4	9.7	11.2	31.2	1.0	1.6	11.9
[45], 4 paths	9.5	–	–	13.3	–	–	6.8	–	–	4.1	–	–
[45], 8 paths	8.4	–	–	11.4	–	–	4.1	–	–	2.7	–	–
[140]	–	9.5	–	–	13.3	–	–	5.9	–	–	3.9	–
[162]	3.5	11.1	**9.6**	7.5	14.7	19.4	**3.6**	**4.2**	14.0	**0.5**	**0.9**	**2.8**
[165]	17.1	25.9	25.8	21.5	28.1	28.8	4.5	6.0	**12.7**	6.0	7.5	18.2
[164]	5.4	11.0	13.9	7.2	12.6	**17.4**	3.8	4.3	14.2	1.2	1.7	5.6
[166]	9.3	11.1	17.5	**6.0**	**7.4**	18.7	8.8	16.4	20.0	3.9	12.0	10.3

Table 5.2: The accuracy of some FPGA-based methods on Middlebury images [35, 139]. The %'s are of pixels with a disparity error > 1 pixel from ground truth: non-occ. = non-occluded pixels only, disc. = pixels near discontinuities only.

Figure 5.5: Qualitative examples of our approach on KITTI frames [41, 169]. Column 1: input left eye image; column 2: ground truth disparities; column 3: the disparities produced by our approach; column 4: error image (blue = low, red = high).

Method	Resolution	Disparities	FPS	Environment
Ours	384x288	32	301	Xilinx ZC706
	450x375	64	198	
	640x480	128	109	
	1242x375	128	72	
[45], 4 paths	640x480	64	66–167	Xilinx Virtex 5
	640x480	128	37–103	
[140]	340x200	64	27	Xilinx Virtex 4
[163]	640x480	32	≥ 30	Xilinx Spartan 6 LX
[162]	352x288	64	1121	Altera Stratix IV
	640x480	64	357	
	1024x768	128	129	
	1920x1080	256	47.6	
[165]	320x240	64	115	Xilinx Virtex 5
	320x240	128	66	
	640x480	64	30	
	800x600	64	19	
[160]	1024x508	128	15	Xilinx Spartan 6
[161]	752x480	32	≥ 60	Xilinx Artix 7
[164]	1024x768	64	60	Altera EP3SL150
[166]	640x480	32	101	Xilinx Zynq-7000

Table 5.3: The frame rates we can achieve, in comparison to those achieved by other FPGA-based methods, for multiple resolution/disparity range combinations.

A justification for the frame rates we achieve is that the embedded system manages to output one output disparity (for both left and right images) per three clock cycles (FPGA clocked at 100 MHz). Although the system is fully pipelined, a strict dependency is incurred due to the use of the energy term of the previous pixel's disparity result for the computation of the current pixel's energy values, as denoted in Equation (5.11). More specifically, the system's bottleneck and upper limit on the frame rate is due to the propagation delay required to compute the minimum of any given pixel's cost vector. This limitation is revisited and described in more detail in the expansion of this work, Section 5.7, where we propose a modification that ultimately sacrifices accuracy for increased speed.

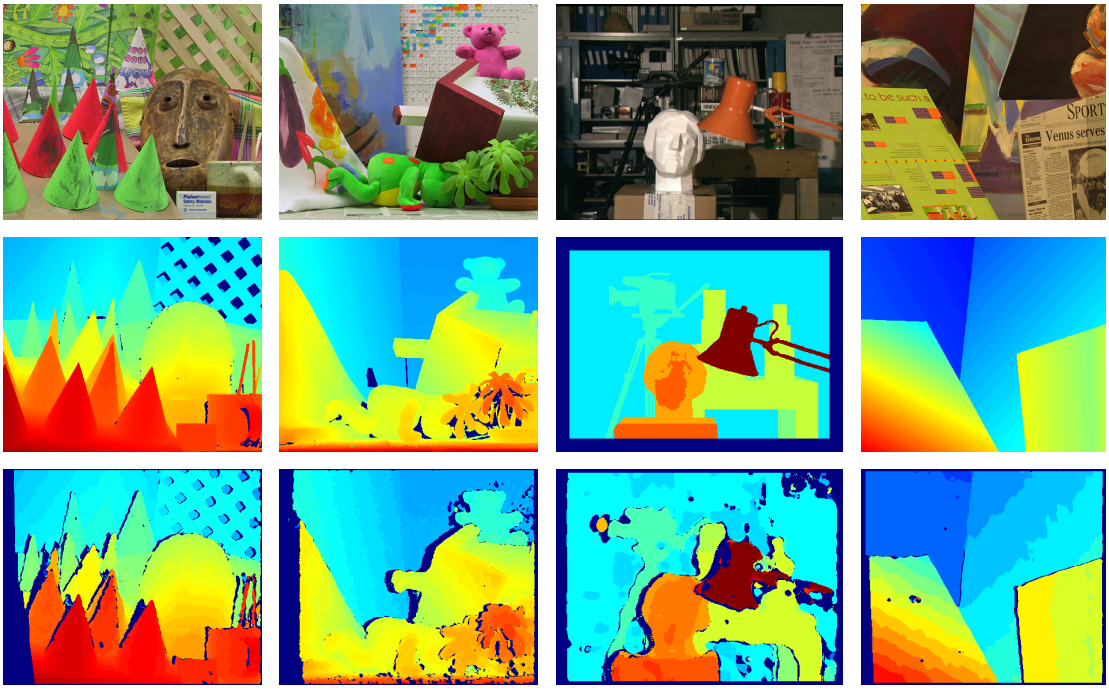


Figure 5.6: Our results on the four most commonly used Middlebury images [35, 139]. Left to right: *Cones*, *Teddy*, *Tsukuba*, *Venus*. Top row: input images; middle row: ground truth disparities; bottom row: the disparities produced by our approach.

Window Width	3	5	7	9	11	13
Frame Rate	72	72	72	72	72	72
Error %	9.3	6.7	5.8	5.4	5.0	4.8
Density %	73	81	83	84	85	85
Interp. Error %	19.4	13.6	12.0	11.1	10.5	9.9
LUT Utilisation %	33.4	37.6	49.9	58.3	67.3	75.7
FF Utilisation %	10.8	14.2	18.1	24.7	32.2	40.5
BRAM Utilisation %	28.9	29.3	29.6	30.0	30.4	30.7
Total FPGA Power (W)	1.68	1.85	2.02	2.36	3.52	3.94

Table 5.4: The impact of varying the Census Transform window size on the frame rate, error and FPGA resource utilization. As the window size increases, the frame rate remains constant, the error on images from the KITTI dataset [41, 169] decreases, and the FPGA resource utilization increases. Power consumption estimates were obtained from Xilinx Vivado.

5.6 Conclusion

In this chapter, we have presented R^{3}SGM, a variant of the well-known Semi-Global Matching (SGM) method [159] for stereo disparity estimation that is better-suited to raster processing on an FPGA. We draw inspiration from the recent More Global Matching work of Facciolo *et al.* [168], which mitigated the streaking artifacts that afflict SGM by

incorporating information from two directions into the costs associated with each scan line. Due to the memory access pattern involved in some of its directional minimizations, MGM proves difficult to efficiently accelerate. Instead, we propose a method that uses only a single, raster-friendly minimization that incorporates information from four directions at once.

Our approach compares favourably with the two state-of-the-art GPU-based methods [150, 176] that can process the KITTI dataset in real time, achieving similar levels of accuracy whilst reducing the power consumption by two orders of magnitude. Moreover, in comparison to other FPGA-based methods on the Middlebury dataset, we achieve comparable accuracy either at a much higher frame rate (c.f. [140, 165]), using simpler, cheaper hardware (c.f. [162]) or handling greater disparity ranges (c.f. [164, 166]). Our approach achieves a state-of-the-art balance between accuracy, power efficiency and speed, making it particularly well suited to real-time applications that require low power consumption, such as prosthetic glasses and micro-UAVs.

5.7 Expansion to the Original Paper

As it stands, R^3SGM is particularly and specifically suited to the in-stream processing style used with FPGA accelerators. It exploits parallelism at the level of the computations involved when processing the output value for a single pixel. Conversely, it does not take advantage of pixel-level parallelism and, in fact, would be a very poor approach to use on a GPU and CPU platform. Therefore, being so specialized as a fast and accurate stereo processing algorithm for FPGA based platforms, it is important that it really does maximize all that it can. Although R^3SGM achieves a good balance between accuracy and speed, there is still room for improvement. Namely, despite generating greater than real-time frame rates for most intents and purposes, it is limited in how much faster it can be.

5.7.1 Speed Bottleneck

As noted in the paper in Section 5.5, the system implementing R^3SGM on the FPGA platform is only able to achieve a throughput of one pixel every three clock cycles (100 MHz clock). This limited throughput is due to dependencies that exist between consecutive pixels and cannot be accelerated by the use of a faster clock. The root of the problem lay with the neighbourhood of pixels that are used to smooth the cost of the current pixel and the fact that pixels are processed in raster order. Referring to Figure 5.7 and Equation 5.4, the cost associated with a given disparity d of pixel $\mathbf{p}$ is obtained by summing the unary cost associated with it together with the smallest of,

- The cost of the neighbouring pixel's disparity at the same index (light blue, Figure 5.7),
- The cost of the neighbouring pixel's disparity at the previous index penalized with a penalty P_1 (light yellow, Figure 5.7),
- The cost of the neighbouring pixel's disparity at the subsequent index penalized with a penalty P_1 (light brown, Figure 5.7)
- The smallest cost contained within the entire cost vector of the neighbouring pixel penalized a penalty P_2 (light green, Figure 5.7)

It is the last of these four options, together with the raster scanning nature of the processing involved that combine to give rise to the problematic bottleneck. More specifically, the problem lies when considering the influence coming from the immediately preceding pixel (to the left of the current pixel). Using the overall minimum across the entirety of its disparity cost vector as one of the options (the green box in Figure 5.7)

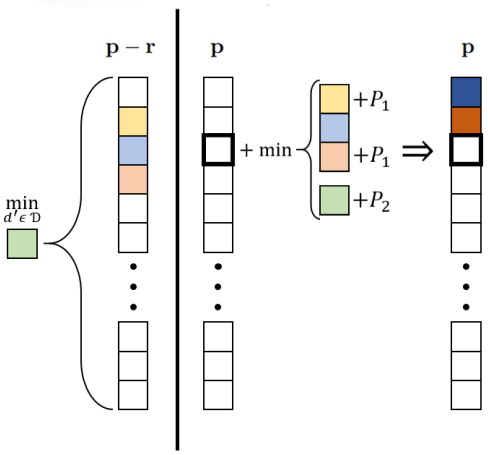


Figure 5.7: Isolating R^3SGM bottleneck

implies that the final result for that pixel is already known. For the purpose of maximizing the throughput to the ideal of one pixel per clock cycle, the computation of a given's pixel result must therefore be known within one clock cycle so that it is available to use in the computations involved with the subsequent pixel. With a 100 MHz clock, this implies that the total available amount of time that the input has to propagate through the logic circuity on the critical path a is 10 ns. In the case of R^3SGM, due to large number of operations involved on the critical path, this is unachievable. Instead, the propagation delay of the critical path is at best within the 20ns-30ns range (varies by configuration, *e.g.* window size). As the result from the immediately previous pixel is required for a subset of the computations, the current pixel must endure that delay before commencing those computations. It does this by providing three 100 MHz clock cycles per pixel computation (within which the delay fits). Increasing the clock frequency does little to increase frame rate as the propagation delay along the critical path is not a function of clock frequency. Although it allows for finer granularity, *e.g.* accounting for a 23 ns delay with five 200 MHz clock periods instead of three 100 MHz clock periods, increasing clock frequency generally only increases the number of clock cycles required to account for a given delay. An additional consequence of this dependency induced delay is the decreased ability to maximize the pipeline-ability of the algorithm. Without the ability to begin computations of neighbouring pixels on immediately subsequent clock cycles due to the implicit need for the former to terminate, the system is limited in the benefits it can get from pipelined operations.

5.7.2 Removing the Left Direction

Unlike the influence from the leftward direction ($\rightarrow$), the cost smoothing from the directions above the current pixel ($\searrow, \downarrow, \swarrow$) are not affected by the same bottleneck. With

these directions of influence, the results are not immediately needed in the computation of the subsequent pixel. Instead, they are stored in line buffers (as explained in Section 5.5), only to be accessed in one "row's worth" of pixels, *i.e.* when the pixel on the row below needs knowledge of the cost vector and minimum (Figure 5.7). For this reason, the same propagation delay that was problematic in the leftward direction is now negligible when compared to the length of time required to process an entire row of pixels. As such, as long as images are wider than $\frac{Propagation\ Delay}{Clock\ Period}$ pixels for a given clock frequency and design configuration, the required results from the relevant pixels above will always be available. With the process from these directions independent in this way, the operations can be separated onto different clocks and efficiently pipelined to have the throughput achieve one pixel per clock cycle.

By only keeping the ($\searrow, \downarrow, \swarrow$) directions and ignoring the influence from the left ($\rightarrow$), it is therefore possible to output depth maps at a much higher frame rate. By no longer having a dependency bottleneck, not only does the system achieve a one pixel per clock cycle throughput - by itself, a threefold gain in speed - but it can now also benefit from increasing clock frequency (whilst maintaining the 1px/clock-cycle throughput). However, removing the leftward direction to only focus on the directions from above will have an impact on accuracy. As illustrated in Figure 5.8, this modification made to R^3SGM greatly decreases the region of influence on any given pixel. Therefore, it is important to weigh whether the increase in the frame rate is worth the cost in accuracy that this approach will have.

5.7.3 Differences in Results

Re-evaluating the modified algorithm on the same images from the KITTI dataset leads to interesting results. First of all, similar to Table 5.1 but without the platform

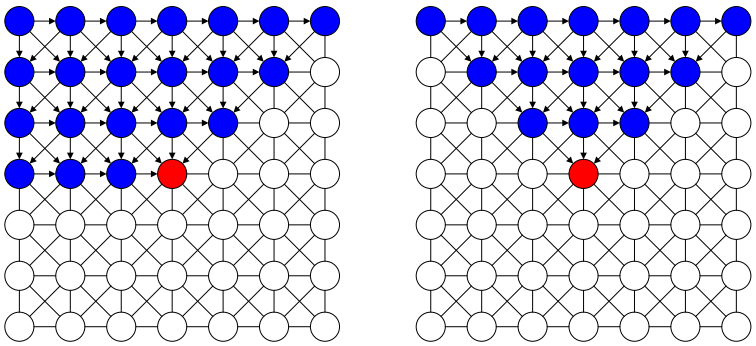


Figure 5.8: Impact of left direction on R^3SGM region of influence. The original region of influence of R^3SGM in (a) compared to the faster implementation of R^3SGM that only considers the directions of influence from above in (b).

Method	D1 Valid	Density	D1 Interpolated	Runtime
R^3SGM - 3 Direction	4.2%	79.0%	10.0%	0.002s
R^3SGM - 4 Direction	4.8%	85.0%	9.9%	0.014s
DeepCostAggr [176]	–	99.98%	6.3%	0.03s
CSCT+SGM+MF [150]	–	100%	8.2%	0.006s

Table 5.5: The quantitative results on the Stereo 2015 subset of the KITTI dataset [41, 169]. D1 Valid: error rate on the pixels surviving the LR check; Density: % of pixels output by the algorithm (following LR check for R^3SGM); D1 Interpolated: error rate after interpolating according to the KITTI protocol. A threshold of 3 disparity values or 5%, whichever is greater (*i.e.* the standard thresholds for KITTI) is used. The removal of the bottleneck stemming from the leftward direction enables the system to operate at a speed six times greater than the original approach. The loss in interpolated accuracy is very small and although the density of the non-interpolated results is notably less, the accuracy over that density is actually slightly better.

information and power consumption (which remained largely unchanged),Table 5.5 highlights foremost the substantially faster frame rates of the 3-Direction version of R^3SGM. As previously explained, this approximately sixfold increase in speed is not only due to the pixel/clock-cycle throughput, but also due to the use of a faster 200 MHz clock instead of 100 MHz clock. On the KITTI dataset, the method is now by far the fastest method listed as well as consuming significantly less power. As shown in Table 5.6, when a comparison is now made to the other FPGA methods across different image resolutions, the new approach is now clearly the fastest. At this throughput, the 3-Direction version of R^3SGM can achieve close to 100 fps for Full HD images (1920×1080), making it more than capable for applications where high resolution depth maps are needed in hard real time. With regards to accuracy, whilst it is true that the diminished region of influence has an effect on the overall accuracy achieved, this effect is not that substantial and not necessarily entirely negative. As shown in Table 5.5, following the interpolation of missing points, the accuracy of the 3-Direction version of R^3SGM is only 0.1% worse than the original. On the other hand, whilst the modified version of R^3SGM also suffers from slightly worse output depth map density (6.0% less), when only considering pixels that have survived the Left-Right consistency check, their accuracy is actually higher than those output from the original algorithm.

5.7.4 Summary

In summary, processing on the FPGA is highly efficient when it can be done in a streaming fashion. In this way, communicating with external memory is minimized and smaller amounts of local memory can be leveraged to buffer the required data from the stream. This approach also allows us to place the FPGA directly in a pipeline in between the

Method	Resolution	Disparities	FPS	Environment
R^3SGM 3-Direction	384x288	32	**1808**	Xilinx ZC706
	450x375	64	**1185**	
	640x480	128	**651**	
	1242x375	128	**429**	
R^3SGM 4-Direction	384x288	32	301	Xilinx ZC706
	450x375	64	198	
	640x480	128	109	
	1242x375	128	72	
[45], 4 paths	640x480	64	66–167	Xilinx Virtex 5
	640x480	128	37–103	
[140]	340x200	64	27	Xilinx Virtex 4
[163]	640x480	32	≥ 30	Xilinx Spartan 6 LX
[162]	352x288	64	1121	Altera Stratix IV
	640x480	64	357	
	1024x768	128	129	
	1920x1080	256	47.6	
[165]	320x240	64	115	Xilinx Virtex 5
	320x240	128	66	
	640x480	64	30	
	800x600	64	19	
[160]	1024x508	128	15	Xilinx Spartan 6
[161]	752x480	32	≥ 60	Xilinx Artix 7
[164]	1024x768	64	60	Altera EP3SL150
[166]	640x480	32	101	Xilinx Zynq-7000

Table 5.6: The frame rates achieved of the modified version of R^3SGM, in comparison to those achieved by other FPGA-based methods and the originally proposed algorithm, for multiple resolution/disparity range combinations. Without the leftward direction causing a bottleneck, the algorithm now achieves significantly faster speeds - a sixfold increase.

image sensors and the host computer where it can immediately process the image data as it is readout by the image sensors (it does not require an entire image to be fully captured and stored in memory). R^3SGM proposes an efficient manner of performing this type of in-stream processing for computing the depth for stereo pairs of images modelled after the well-established SGM approach. Although it obtains results that are competitive with SGM whilst requiring substantially less memory and operating a real-time frame rates, it suffers from a dependency that limits the speed it is able to achieve. This dependency is related to the raster scan order in which the pixels are processed and the smoothing

influence that originates from the same direction. As a pixel depends on the immediately previously computed pixel's output value throughout its own computations, it cannot commence before that previous pixel has finished being processed. Being a recursive process, this forces the system to adhere and wait for the propagation delay along the critical path of that computation. This not only locks the upper limit on speed to this delay, but impacts the amount of pipelining that can be taken advantage of.

Unlike the leftward direction, the directions from above are not immediately needed after they are computed and are instead stored for use at a much later time. By focusing solely on the influence from these directions and with all the necessary data available in local memory, the new system is no longer stalled by dependencies and much more amenable to pipelining. The result is a system that gains a sixfold increase in speed, achieving one pixel per clock cycle throughput with a 200 MHz clock. However, as the algorithm's region of influence for smoothing is decreased, there is a small decrease in overall accuracy (when interpolating missing values) as well as output image density.

Despite the decrease in accuracy and density, the substantial gain in throughput and overall frame rate of this modified approach makes it a worthwhile alternative to consider. In addition, although the density of the output image does suffer, the accuracy over the pixels within that smaller set is slightly improved. Such an approach may therefore be preferred in low power systems that favour higher frame rates and where sparse depth images are sufficient or where the fast processing time is taken advantage and used to generate initial approximations of depth maps that are subsequently post-processed and utilized to construct a denser image. With the ability to generate rapid first approximations, more time can be dedicated to more complex but more difficult to accelerate processes whilst retaining real-time speeds.

For the great doesn't happen through impulse alone, and is a succession of little things that are brought together.

— Vincent Van Gogh

6

Combining Priors for More Accurate Real-Time Depth Processing

Contents

6.1 Preface 90
6.2 Introduction 91
6.3 Method 92
6.3.1 Sparse Disparity Computation 94
6.3.2 Combined Disparity Optimization 99
6.4 Results 101
6.5 Conclusions 102

6.1 Preface

In the previous chapter, we achieved faster than real-time results with R^{3}SGM on only the FPGA portion of the ZC706. When considering the modified version elaborated upon in Section 5.7, the approach meets both the criteria of speed and power consumption for many mobile, low-power real-time systems. In addition, compared to the simple SAD approach developed in chapter 3, the effects of the disparity cost smoothing as well as the use of median filters and LR checking contribute to significant gains in accuracy. Although, as shown in Table 6.1, R^{3}SGM consumes more power due to the additional FPGA logic requirement, the trade-off for the additional accuracy warrants it.

	Window Width	5	7	9	11	13	15
SAD	**Error %**	46.60	39.01	34.39	31.38	29.35	28.01
	Power (W)	0.67	0.60	0.86	0.83	0.86	1.40
R^{3}SGM	**Error %**	13.6	12.0	11.1	10.5	9.9	N/A
	Power (W)	1.85	2.02	2.36	3.52	3.94	N/A

Table 6.1: Comparing the accuracy of the simple sum of absolute differences (SAD) approach to that obtained with R^{3}SGM. The accuracy is measured over the same set of images from the KITTI 2015 dataset and compared across the different window sizes used for feature matching (SAD & CT).

One disadvantage is that, unlike the SAD, the inter-pixel dependency present in the R^{3}SGM approaches removes the ability to split the stereo image processing among multiple accelerators (each computing a separate subset of the image pair) to further speed up the frame rate. Nevertheless, with the frame rates of the modified R^{3}SGM surpassing real-time requirements on even HD resolution images, this difference is unlikely to be important in most applications.

In practice, the excess in speed opens the opportunity for further improvement in accuracy by increasing the complexity and quantity of computations involved in the processing of the stereo image pair. This deepening of the stereo image processing pipeline is precisely what is developed in this chapter.

This chapter reflects the work published in "Real-time Highly Accurate Dense Depth on a Power Budget using an FPGA-CPU Hybrid SoC" [177] with myself as the primary author. Although the contents of this chapter are largely identical to the originally published journal paper, a few changes have been made to add additional detail, improve clarity and improve structure. This includes the addition of two new figures. The introduction has been shortened as some of the content was redundant with that of the main introduction at the beginning this thesis. Lastly, mentions to other works that have been included in this thesis have been altered so that they point to locations in the thesis rather than to the relevant papers.

6.2 Introduction

Two popular solutions offering a good trade-off between speed and accuracy are Semi-Global Matching (SGM) [159] and ELAS [86]. SGM computes initial matching hypotheses by comparing patches around pixels in the left and right images, then approximates a costly image-wide smoothness constraint with the sum of several directional minimizations over the disparity range. By contrast, ELAS first identifies a set of sparse but reliable correspondences to provide a coarse approximation of the scene geometry, then uses them to define slanted plane priors that guide the final dense matching stage. We propose a novel stereo pipeline that efficiently combines the predictions of these two algorithms, achieving high accuracy and overcoming some of the limitations of each algorithm. First, we use multiple passes of a fast SGM variant [151], left-right consistency checking and decimation to obtain a sparse but reliable set of correspondences. Then, we use these as the support points for ELAS to obtain disparity priors from slanted planes. Finally, we incorporate these disparity priors into a final SGM-based optimization (again based on [151]) to achieve dense predictions with high accuracy.

Our pipeline targets not only accuracy, but also speed, aiming for real-time execution (30 fps) on an embedded platform. Recent works have deployed SGM successfully in real time both on multi-core CPUs [44] and GPUs [46, 150], but in real-world scenarios, power constraints often force us to rely on low-power devices like FPGAs. The development of reliable stereo pipelines for FPGAs is an active research field [85, 123, 178–183], with recent works proposing FPGA-friendly variants of SGM [45, 140, 143, 150, 161, 163] or ELAS [126]. However, FPGA implementations of stereo algorithms usually perform some kind of approximation to deal with the limited resources available and to traverse the pixels in raster order.

We show how some of the intrinsic limitations of a pure FPGA-based implementation can be mitigated by appropriately leveraging a new-generation hybrid system on a chip (SoC), e.g. the Xilinx ZCU104, which combines both an ARM processor and an FPGA, with shared direct memory access, into a single chip. Recently, several works have explored the deployment of stereo methods on such platforms: both [161] and [179] use the CPU mainly for handling communication and controlling peripherals, while [126] actively leverages the CPU to execute iterative steps that would be infeasible on an FPGA (e.g. Delaunay triangulation). Similar to [126], we propose to actively use the elaboration capability of the built-in CPU to handle I/O and to execute part of the ELAS pipeline, while deploying all the other elaboration blocks on the FPGA. We show how our pipeline outperforms previously published works by achieving an 8.7% error rate on the challenging KITTI 2015 dataset [40, 41], while still operating with real-time performance and low power consumption.

6.3 Method

Our overall pipeline is shown in Figure 6.1. It consists of several different components which we describe in the subsections that follow. The system leverages both parts of the FPGA-CPU hybrid SoC to achieve optimal results. Tasks that are very data intense, but which access that data in a predictable manner, are run on dedicated FPGA accelerators to benefit from their parallel processing capability. In addition, they can take advantage of the FPGA accelerators' internal ability to pipeline data so that multiple inputs are processed together in staggered fashion. Tasks that are very dynamic and unpredictable, which often involve many unforeseen or random accesses to external memory, are run on the CPU since they benefit both from the significantly faster clock frequency of the CPU as well as its ability to access memory in constant time (CPU memory accesses can be sped up via appropriate use of the cache). To minimize the amount of FPGA resources used by our method, as well as allow the deployment of the design on a real platform, we reuse some accelerators whilst buffering intermediate results in RAM. We will detail which blocks are reused in our final design in the rest of this section.

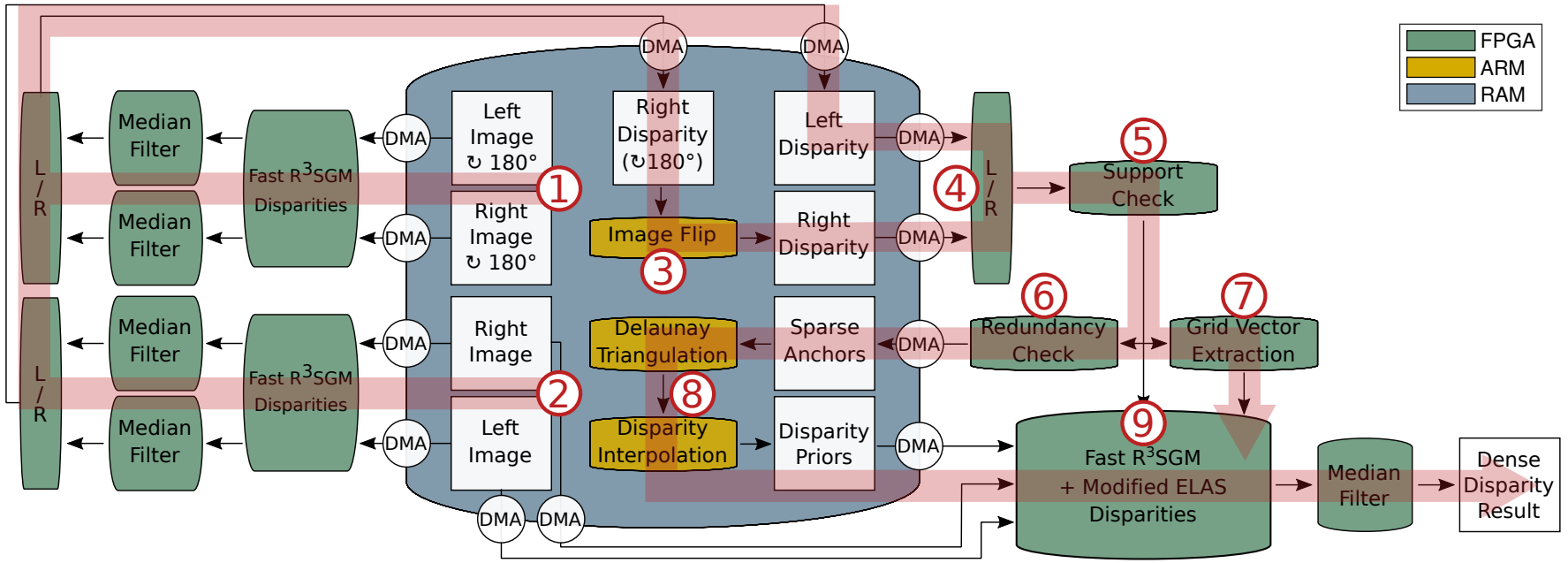


Figure 6.1: Overview of our approach. First, we use Fast R^{3}SGM (see §6.3.1) to compute disparity images for the input stereo pair (in raster (1) and reverse-raster (2) order). We then flip the right result (3) and perform a left-right consistency check (4) to obtain an accurate but sparse disparity map for the left input image (see §6.3.1). Next, as ELAS [86] does, we perform *support checking* (5) (see §6.3.1) to remove points whose disparities appear abnormal relative to neighbouring pixels: this yields a sparser support point image that contains only points with confident disparities. This support point image is subsequently used in multiple ways: (i) it is further sparsified via a *redundancy check* (6), producing sparse anchors that are then used to generate plane disparity priors through a *triangulation* and *interpolation* process (8) (see §6.3.1); (ii) it is split into a grid (7) where, for each grid cell, a binary vector representing the set of viable disparities is computed (see §6.3.1). Finally, the support point image is combined with the outputs of (i) and (ii) in a *disparity optimization* (9) that combines R^{3}SGM and ELAS to produce a dense disparity image (see §6.3.2). We then median filter this image for robustness to produce the final result.

6.3.1 Sparse Disparity Computation

Fast R^{3}SGM (① & ②)

Initially, we use a modified version of R^{3}SGM [151] (a memory-efficient adaptation of classic SGM [159] to FPGAs), which we call Fast R^{3}SGM, to compute disparity images for input stereo pairs. The original version of R^{3}SGM aggregated contributions to the disparity of each pixel along four different scanlines: three above the pixel, and one to the left. As mentioned in [151] (and detailed in this thesis starting on p. 83), using the left scanline severely limits the overall throughput of the system (one disparity value is output every three clock cycles) due to a blocking dependency between immediately successive pixels. To avoid this, we modify the approach to use only the scanlines above the pixel, allowing us to output one disparity per clock cycle. The mild loss in accuracy this causes is more than compensated for by the improvements yielded by the rest of our pipeline.

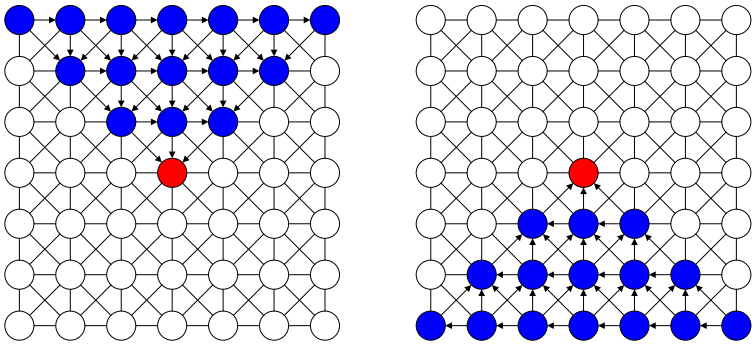


Figure 6.2: Comparing the implicit biases that exist in raster and reverse-raster passes of R^{3}SGM. The region of influence used in the original version of Fast R^{3}SGM contains only pixels above the pixel of interest (a), and as such the disparity value computed for that pixel is unaffected by the pixels below it. The opposite holds true when performing Fast R^{3}SGM in reverse-raster order (b).

We process each input pair twice: once in raster order (①), and once in reverse-raster order (②), yielding two disparity images overall. The advantage of this is that, as illustrated in Figure 6.2, the raster and reverse-raster passes of R^{3}SGM will base the disparity for each pixel on the disparities of pixels in different regions of influence. By comparing the results output by both these separate passes, we can identify pixels for which the difference in bias caused a significant change in the resulting output value. These can then be removed so that only pixels that maintain the same value, regardless of bias, are retained. Through this type of consistency check, the confidence and accuracy of the results can be improved.

In our implementation, we deploy a single instance of the Fast R^{3}SGM block, together with the associated median filtering and L/R consistency checking blocks. We first

feed the blocks with the raster-order stereo pair (①), then with the reverse-raster-order pair (②), storing the disparities resulting from each pair back into RAM between the computations. Example images resulting from this two-pass process are depicted in Figure 6.3. For further architectural details of the internal structure of the main blocks, we refer the reader back to Chapter 5.

Consolidating Consistency Checking (③ & ④)

Each pass of Fast R^{3}SGM outputs a disparity map that has been checked for consistency using the first input as the reference image [151]. The raster pass outputs a disparity map for the left input image; the reverse-raster pass outputs one for the (reversed) right input image. Due to the streaming nature of the disparity computation, however, the results suffer from a raster or reverse-raster scan bias, i.e. the disparity value of any given pixel

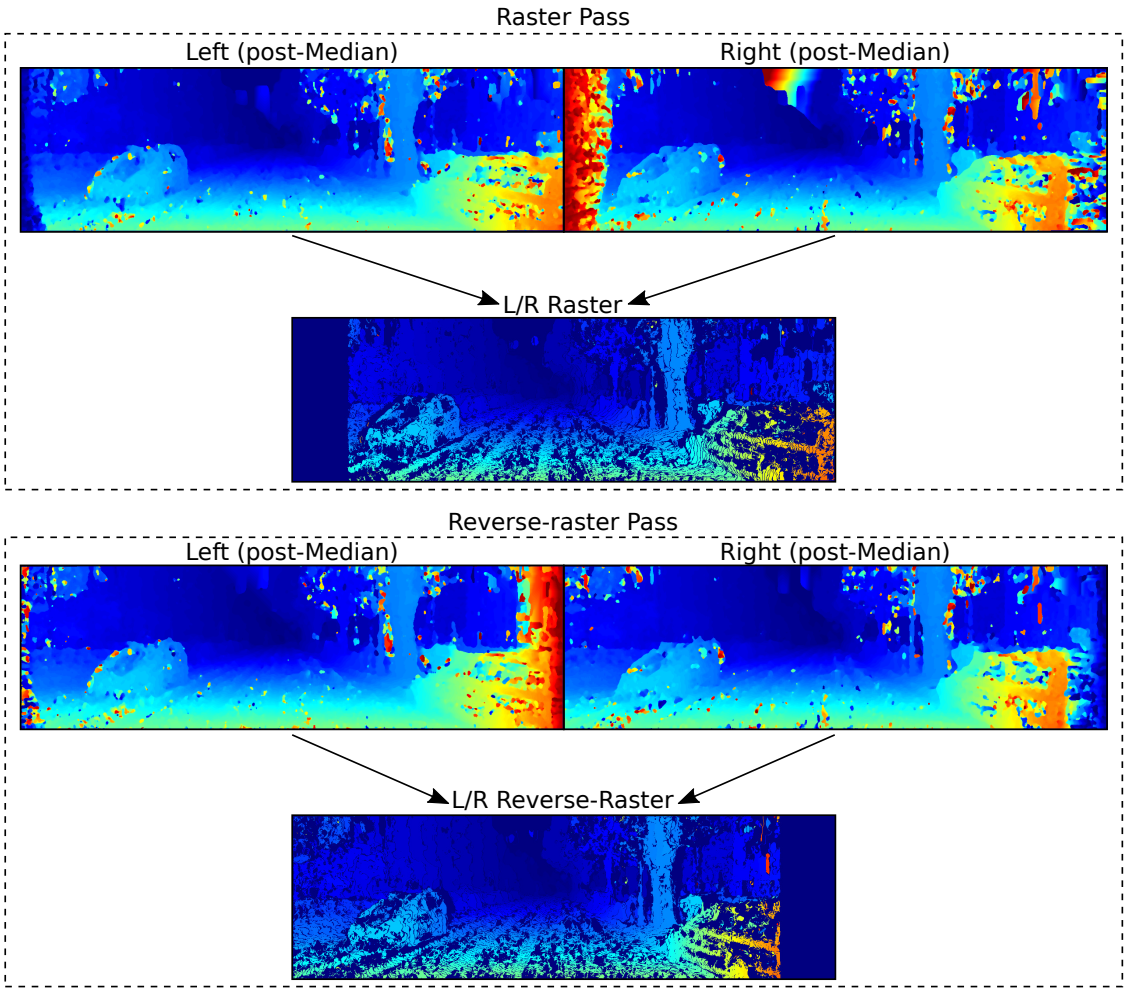


Figure 6.3: An example showing the effects of performing L/R consistency checks after the initial raster and reverse-raster passes of Fast R^{3}SGM and the subsequent median filtering. Note that in the reverse-raster case, we have flipped the images for easier comparison with the raster case.

is encouraged to be similar to those computed before it. To reconcile the inconsistencies between these two disparity maps, we perform a further left-right consistency check (④), which yields an accurate but sparse disparity map for the left input image as its result (see Figure 6.1). The memory access pattern of such a process is problematic, though, as the first pixels in the left disparity map need to be checked against the last pixels in the right disparity map. To overcome this problem, we first reverse the latter image on the CPU (③) (since this is an inherently sequential process, it benefits from the higher clock rate provided by the ARM core), then perform a standard left-right consistency check (④) (on the programmable logic).

Generation of Priors (⑤, ⑥, ⑦ & ⑧)

Using the sparse disparity map output by the consolidating consistency check, we adapt the ELAS method described in [86] to generate priors that can be fed into a combined disparity optimization process (see §6.3.2) to produce a more accurate and dense final result. The prior generation process begins by taking the disparity map produced by §6.3.1 as input and producing a support point image (see §6.3.1) containing sparse but confident disparities (⑤). The support points are then fed to two more blocks (⑥ + ⑧, & (⑦) before being used by the final disparity optimization process: (i) a redundancy checking and disparity prior generation block, which first computes a sparse anchor point image (⑥) and then triangulates such anchors to generate disparity priors for all pixels in the image (⑧) (see §6.3.1); and (ii) a grid vector extraction block (⑦)that divides the support points image into a grid and then determines the set of possible disparities for each cell (see §6.3.1).

Support Checking (⑤)

To produce the support point image (Figure 6.4), we filter the sparse disparity map to remove any pixels whose disparities are not sufficiently supported by the pixels in their immediate neighbourhoods (in practice, a square window centred on each pixel). For a pixel to be considered "supported", there must exist, in its neighbourhood, another predefined number of pixels that have very similar disparity values (e.g. at least 10 pixels within a 5×5 window that differ by less than 5 from that of the centre pixel). The disparities of all other pixels are marked as invalid. The resulting support point image will evidently be sparser than the original disparity map, since we have kept only those pixels about whose disparities we can be reasonably confident.

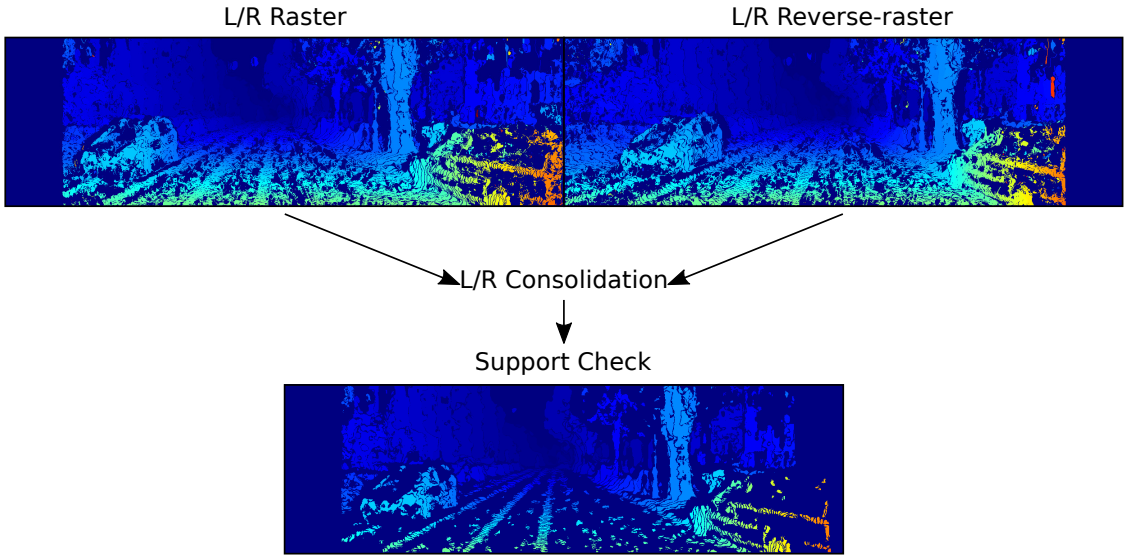


Figure 6.4: The results of performing a consolidating consistency check and subsequent support check on the raster and reverse-raster outputs from Figure 6.3.

Redundancy Checking and Disparity Prior Generation (⑥ & ⑧)

To produce the anchor image (see Figure 6.5), we further sparsify the support point image from §6.3.1 by processing it in raster order and invalidating any pixel whose disparity has already been seen within a window behind and above the pixel (⑥). Unlike in Chapter 4, where each pixel (x,y) used a window of

$$\{(x,y-\delta_y) : 0 < \delta_y \leq 2K\} \cup \{(x-\delta_x,y) : 0 < \delta_x \leq 2K\},$$

where K was set to 5, which only encompassed points in the same row or same column as the pixel being processed, here we use a larger window of

$$\begin{array}{ll} & \{(x+\delta_x,y+\delta_y) : -K \leq \delta_x \leq K, -2K \leq \delta_y < 0\} \\ \cup & \{(x-\delta_x,y) : 0 < \delta_x \leq K\}. \end{array}$$

This has the effect of creating a sparser anchor image than that used in Chapter 4, significantly speeding up the subsequent Delaunay triangulation process. Whilst this inevitably reduces the granularity of the generated triangles, its impact on the quality of the subsequent depth priors is minor, as was shown in that same chapter. As the Delaunay triangulation process was additionally shown to be a key bottleneck, the advantage of reducing the number of points to triangulate (thus reducing CPU processing time) outweighs the marginal benefit in accuracy obtained with more fine-grained triangles.

Finally, to produce the disparity priors, we first move the anchor points image back to RAM, then perform a Delaunay triangulation of those points, and finally compute the disparity of each non-anchor point located within one of the Delaunay triangles by interpolating the disparities of the triangle's vertices (⑧) (depicted in Figure 6.6). The

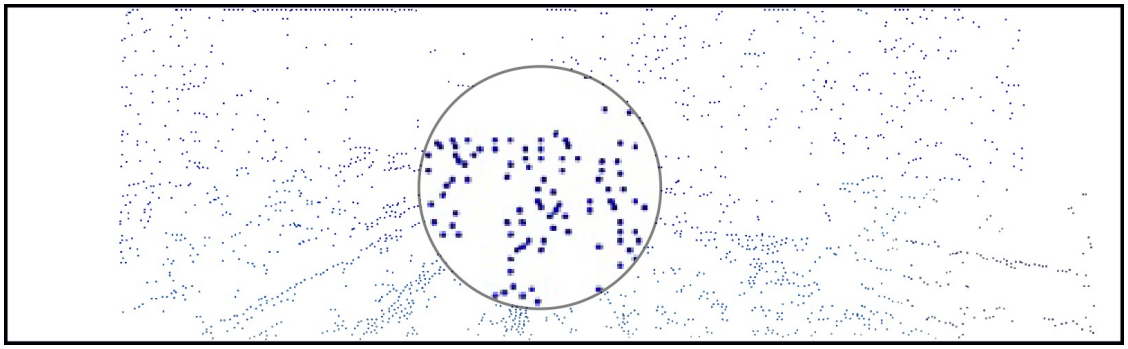


Figure 6.5: The results of performing a redundancy check on the support point image from Figure 6.4. Note the sparseness of the anchor points, which significantly speeds up the subsequent Delaunay triangulation process. A portion of the image has been magnified for better visibility.

entirety of this process is performed by the CPU, since the triangulation and interpolation algorithms are inherently non-sequential in their memory access patterns, and can benefit from both the availability of memory caches and the higher speed of the ARM core.

Grid Vector Extraction (⑦)

The final input to the combined disparity optimization we describe in §6.3.2 is a set of binary *grid vectors* used to determine which disparities are suitable for each part of the image. To produce such vectors (⑦), we first divide the support point image into a regular grid (with cells of size 50×50 in our implementation). Then, for each cell, we find the valid disparity values within it, and store both those and their neighbouring disparities (± 1) into a binary grid vector for that cell. Refer back to Chapter 4 for more details.

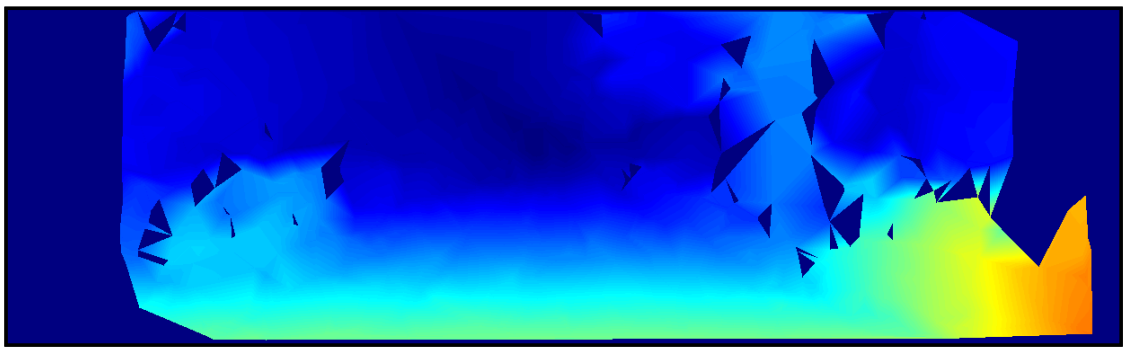


Figure 6.6: The plane priors produced by constructing a Delaunay triangulation based on the sparse anchor points in Figure 6.5, and then linearly interpolating the disparities within each triangle.

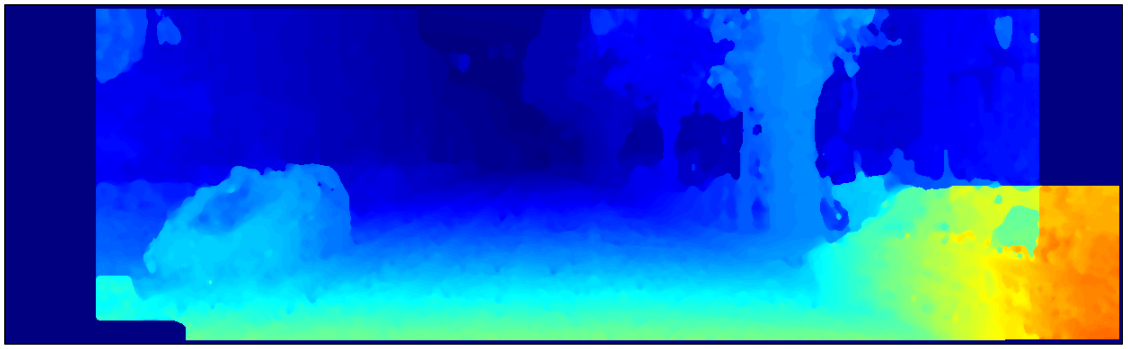


Figure 6.7: The final dense disparity image for our running example, produced using the combined disparity optimisation described in Section 6.3.2.

6.3.2 Combined Disparity Optimization

Finally, we perform a combined disparity optimization (9) that takes into account not only the original pair of input images, but also the plane priors, grid vectors and support points. Essentially, we perform Fast R^{3}SGM, as in §6.3.1 (once again reusing the corresponding FPGA block), but first modifying the cost vectors of the pixels to take the various different priors we have available into account.

The disparities of the support points are fixed throughout and not recomputed. Every cost vector element for a support point (each of which corresponds to a specific disparity) is set to a large arbitrary value, except for the element corresponding to the disparity of the support point, which is set to zero instead. Through the Fast R^{3}SGM smoothing process, pixels near the support point will then naturally be encouraged to adopt disparities similar to that of the support point itself, with the influence of this effect attenuating with distance. To take the disparity prior for each pixel into account, we decrease those elements of its cost vector that correspond to disparities close to the prior (more specifically, we superimpose a negative Gaussian over the cost vector, centred on the prior, and decrease the relevant elements within a certain radius accordingly). To make use of the grid vectors, we set all elements of the cost vectors for the pixels within each grid cell that do not appear in the grid vector for that cell to an arbitrarily large value, thus strongly encouraging them not to be selected. As with the effects of the support points, these cost vector modifications are similarly propagated by the Fast R^{3}SGM smoothing process. At the end of this process, we perform a final median filter on the Fast R^{3}SGM result to further mitigate impulsive noise, ultimately yielding a dense, accurate disparity map as demonstrated in Figure 6.7.

Method	Background	Foreground	All	Density	Runtime (s)	Environment	Power Used (W)
Ours	7.2%	17.3%	8.7%	99.7%	0.019	FPGA (Xilinx ZCU104)	≈ 5
R^3SGM [151]	–	–	9.9%	85.0%	0.014	FPGA (Xilinx ZC706)	≈ 4
ELAS-FPGA [126]	–	–	13.6%	–	0.095	FPGA (Xilinx ZC706)	≈ 3
DeepCostAggr [176]	5.3%	11.4%	6.3%	99.98%	0.03	Nvidia GTX Titan X	≈ 250
CSCT+SGM+MF [150]	6.9%	14.9%	8.2%	100%	0.006	Nvidia GTX Titan X	≈ 250

Table 6.2: The quantitative results of our approach, in comparison to state-of-the-art real-time methods, on the Stereo 2015 subset of the KITTI benchmark [40, 41]. As in the official evaluation protocol, we report the percentage of accurate disparities (using a threshold of < 3 disparity values or 5%, whichever is greater) after an interpolation step (meant to assign a disparity value to all pixels in the image), on respectively the subsets of background, foreground and all pixels. We additionally report the density of valid disparity values. As can be seen, with the exception of R^3SGM [151], all methods provide almost dense disparity images, therefore the extra interpolation step mandated by the benchmark is not strictly required to obtain usable disparity images. Finally, for each method, we report the typical time required to process a stereo pair, as well as the approximate power consumption of the platform used. Whilst all approaches can process images in real time, only the FPGA-based methods (ours and [151]) can do so in a power-efficient manner, with ours providing $\approx 12\%$ additional accuracy and much higher density w.r.t. [151], at the expense of slightly higher power usage and processing time.

6.4 Results

We developed the FPGA accelerators using the Vivado High-Level Synthesis (HLS) tool, as this approach was quicker, and allowed for greater flexibility and reusability of the components. We deployed the system on a Xilinx ZCU104 board, and all of the power consumption results that we present for our method were estimated by the Xilinx Vivado tool. Although the values provided by the tool are only approximations, they still provide an accurate sense of the power requirements.

We quantitatively evaluate the disparities produced by our approach on the standard KITTI 2015 stereo benchmark [40, 41]. In Table 6.2, we report the average percentages of pixel disparities estimated correctly for background, foreground and all pixels, respectively. We also report average runtimes and power consumptions for both our and alternative methods that achieve real-time processing speeds on the images used in the benchmark (which have a resolution of 1242×375). Whilst the proposed method results in slightly less accurate disparities than the DeepCostAggr [176] and CSCT-SGM-MF [150] methods, it is worth pointing out that both [150, 176] rely on powerful GPUs to achieve real-time processing speed, whereas our approach does so in a much more power-efficient manner, relying only on a hybrid FPGA-CPU board. We also compare favourably to R^3SGM [151], the underlying method on which we base our approach for the estimation of the initial disparities (see §6.3.1), providing more accurate and denser results at a similar speed and with similar power consumption. We similarly outperform the FPGA variant of ELAS [126], achieving a lower error rate at a much higher speed, and with similarly low power consumption.

	Ours	[151]	[126]
Platform	ZCU104	ZC706	ZC706
LUT Utilisation (% / #)	87.5 / ∼202K	75.7 / ∼165K	37.3 / ∼82K
FF Utilisation (% / #)	24.1 / ∼111K	40.5 / ∼177K	22.6 / ∼99K
BRAM Utilisation (% / #)	70.7 / ∼440	30.4 / ∼330	11.9 / ∼130
FPGA Power Usage (W)	≈ 2	3.94	1.21
CPU Power Usage (W)	≈ 3	–	1.7
Total Power Usage (W)	≈ 5	–	2.91
CPU Frequency (MHz)	1200 (4 Cores)	–	800 (2 Cores)
FPGA Frequency (MHz)	200	100	100

Table 6.3: Resources (programmable logic units), power (as estimated by the Xilinx Vivado tool) and clock frequencies used by the proposed approach, when deployed on a Xilinx ZCU104, in comparison to the FPGA-based methods from which we draw inspiration.

In Table 6.3, we detail the hardware resources used by our approach when deployed on our Xilinx ZCU104 board. We break down the amount of logic resources used in the FPGA chip, as well as the power consumption of both the programmable logic and the ARM core. We also report the amount of resource and power used by the methods from which we draw inspiration [126, 151]. Notably, despite making full use of many of the logic resources available on the FPGA, our power consumption remains very low. More specifically, breaking down the resource utilization of the programmable logic amongst the different accelerators, the largest share is taken by the Fast R^3SGM block which, alone, consumes about 65% of the FPGA power. The next most resource-heavy blocks are the ones which perform the median filtering of the disparities, which require approximately 30% of the power. The remaining blocks have much smaller resource requirements, which altogether account for the remaining 5% of the power.

6.5 Conclusions

In this chapter, we have presented a novel approach to computing depth from stereo images on a hybrid FPGA-CPU chip. Our approach uses an adapted version of ELAS [86] to refine the initial sparse disparity map produced by a fast variant of R^3SGM [151], and achieves an 8.7% error rate on the challenging KITTI 2015 dataset [40, 41]. By fully leveraging the capabilities of our hybrid board, we are able to produce highly accurate dense depth at over 50 fps, with a power consumption of only 5W, making our approach attractive for applications in mobile, real-time computing.

There is nothing noble in being superior to your fellow men. True nobility lies in being superior to your former self.

— Ernest Hemingway

7 Conclusion

This thesis began by highlighting that there exists not only a need for a reliable approach to the computation of depth from stereo images, but that this need often has additional constraints. Most notably, in many mobile and lightweight applications, power consumption is a significant factor when choosing which approach to use. Similarly, many systems have strict requirements on the speed and latency with which the depth information is obtained, as well as needing to consider other factors such as size and weight of the platform. Although improvements in stereo image processing have been continuously made since research into the field began in the late 1980s, most of the focus of this research has been on accuracy. An overview of the various computational platforms that currently existed showed that for the purpose of mobile, real-time and low-power computing, FPGAs were justifiably the best suited devices.

The first step in developing a fast stereo algorithm was to see what upper bound could be achieved, through the use of the simplest and most well-established approach, namely SAD block matching. The results we obtained with this approach proved exceptionally fast, being able to achieve a throughput of one pixel per clock cycle and with very low power consumption. Despite the achievement in speed, the accuracy of this approach was not very competitive when compared with more advanced methods.

Following this first step, a new investigation of the literature led to the discovery of an algorithm that touted the ability to quickly process large-resolution stereo images. This algorithm, ELAS, uses two passes over the stereo pair, where the first pass establishes a set of confident correspondences that are then used to establish a coarse representation of the scene to guide the matching process in the subsequent step. Adapting the algorithm to a heterogeneous CPU-FPGA platform proved challenging and required

many changes to adequately accelerate it in the pursuit of speed. This included examining the computational nature of the different parts of the algorithm, distributing the workload efficiently across both FPGA and CPU and restructuring the format and flow of data so as to accommodate the limitations of the platform. The results that were achieved from the final system were mixed, with the FPGA portions of the algorithm excelling and vastly surpassing the results originally reported by ELAS, but with the portions computed on the onboard CPU taking longer due to the relatively less capable chip. The key takeaway from the process was that an algorithm for FPGA acceleration needs to be designed from the ground-up, with the advantages and limitations of FPGA processing being considered.

Returning to the fundamentals of stereo imaging algorithms that have been accelerated with FPGAs, we re-examined the most well-established algorithm, SGM [159]. A newer modification on the algorithm, named MGM [168], expanded on the results achieved by SGM by broadening the region of influence acted upon during the smoothing process of the disparity cost vector of each pixel in the image. Despite better accuracy, MGM had an even greater memory requirement than SGM, and was even more computationally intense. By drawing inspiration from some of the key ideas of MGM, whilst preserving the ability to operate in real time on an FPGA, we proposed R^3SGM, a real-time, raster-respecting variant of semi - and more-than-semi - global matching algorithm. This new algorithm extends the dynamism inherent in the two prior approaches, but keeps the problem tractable by only considering directions of minimization that involve data that received in raster order. The algorithm achieved very similar levels of accuracy to the original SGM, whilst requiring significantly less memory and fewer FPGA resources, and being able to operate in real time. The pursuit of even faster throughput with R^3SGM led to resolving a bottleneck due to propagation delay. This new, faster variant had one less direction of minimization, but the small loss in accuracy that this caused (0.1%) was well worth the threefold increase in the frame rate, *i.e.* the FPGA accelerator was able to achieve a throughput of one pixel per clock cycle.

Armed with this new resource efficient algorithm that was able to produce acceptable depth at very high frame rates, we then attempted to up the accuracy by deepening the stereo imaging pipeline. Inspired by the multi-step approach proposed in ELAS and drawing on the experience obtained by adapting it to the embedded FPGA platform, a new multi-stage algorithm was proposed that combined coarse plane priors with disparity vector smoothing. To balance the raster scan bias introduced by a single pass of R^3SGM, we proposed using a first stage where the same image pair is passed twice through the block; once so that it produces a raster-biased image, and once with the images flipped to produce reverse-raster-biased images. Through a subsequent process of confidence checking and efficient data reduction, a subset of points is kept to produce coarse plane

priors in an ELAS-like fashion as well as to guide, in a more heavy-handed manner, the smoothing process of another R^3SGM pass. The result of this combined disparity optimization proved capable of producing real-time depth images, with very low power consumption and with an accuracy bettering the other approaches.

An important feature of the work done in this thesis was the use of high-level synthesis. Compared to traditional hardware design with VHDL or Verilog, this provided a much more rapid and iteratable approach to the design of FPGA accelerators. This approach to the design of FPGA hardware accelerators may be controversial with regards to the ultimate efficiency and performance that can be achieved, but the experience of using it throughout the course of the research undertaken as part of this thesis has shown that it allowed for much quicker turnaround and flexibility in the designs.

The outcome from this research paves the way for using FPGA based hardware accelerators as platforms for systems where power consumption, hard real-time performance, small form factor and weight as well as low cost are important constraints that need to be balanced. For higher volume production, the outcomes from this research can be equally applicable in the design of specialized and structured ASICs for further cost savings and reductions in power consumption.

Building atop of the research carried out in this thesis, many avenues of new research are possible. Some of the hardware resource utilization by other blocks that we noted were far greater than expected. The median filter, for example, occupied a significant portion of FPGA resource - almost accounting for half the resource as noted in Chapter 6 - despite being a relatively secondary part of the pipeline. This operation may be approximated by a cascade of alternating line medians which would prove much more resource efficient and scale linearly instead of quadratically in the required logic. Similarly, more work could be done to develop alternative pipelines that have additional passes over the stereo images, but that maintain real-time capabilities by focusing more heavily on FPGA accelerators and reducing the number of processes that need to be computed by a CPU. By iteratively improving the depth maps from the same stereo image pair, this type of focus could perhaps prove more accurate and yet faster - despite the additional passes - than one that relied on CPU-dependent priors. Similarly, with the recent rise of deep learning in computer vision, integrating a lightweight neural network into the stereo pipeline could provide much better gains in accuracy whilst remaining real time. Furthermore, future directions of research could involve fusing different modalities of depth perception, *e.g.* LIDAR and stereo-vision, and retaining real-time speed with the help of FPGA accelerators. This could allow for both better accuracy and additional redundancy by taking advantage of the strengths of both modalities.

Lastly, the future of application-focused computer vision lies in the co-design of algorithms with hardware. Without this cooperative approach, benchmark-leading algorithms focusing on a single metric will be doomed to exist solely in the realm of the theoretical. This not only extends to the hardware needed for processing, but to hardware that is relied on throughout the system, including sensors, actuators, *etc*. For example, designing algorithms to take advantages of the strengths of new imaging devices such as event cameras [184, 185] or light field cameras [186, 187] holds the promise of more specific and efficient solutions for many systems.

A wise man proportions his belief to the evidence.

— David Hume

References

[1] Oliver Sacks. *The mind's eye*. London: Picador, 2011.

[2] A. David Milner and Melvyn A. Goodale. *The Visual Brain in Action (Oxford Psychology Series)*. Oxford University Press, 1996.

[3] Helen Oleynikova, Dominik Honegger, and Marc Pollefeys. "Reactive avoidance using embedded stereo vision for MAV flight". In: *ICRA*. 2015.

[4] Pascal Steingrube, Stefan K. Gehrig, and Uwe Franke. "Performance Evaluation of Stereo Algorithms for Automotive Applications". In: *Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems*. ICVS '09. Liège, Belgium: Springer-Verlag, 2009, pp. 285–294.

[5] N. A. Zainuddin, Y. M. Mustafah, Y. A. M. Shawgi, and N. K. A. M. Rashid. "Autonomous Navigation of Mobile Robot Using Kinect Sensor". In: *International Conference on Computer and Communication Engineering*. 2014.

[6] Hans P. Moravec. "The Stanford Cart and the CMU Rover". In: *Autonomous Robot Vehicles*. Ed. by Ingemar J. Cox and Gordon T. Wilfong. New York, NY: Springer New York, 1990, pp. 407–419.

[7] Stuart Golodetz*, Tommaso Cavallari*, Nicholas A Lord*, Victor A Prisacariu, David W Murray, and Philip H S Torr. "Collaborative Large-Scale Dense 3D Reconstruction with Online Inter-Agent Pose Optimisation". In: *TVCG* 24.11 (2018).

[8] Olaf Kähler, Victor Adrian Prisacariu, Carl Yuheng Ren, Xin Sun, Philip Torr, and David Murray. "Very high frame rate volumetric integration of depth images on mobile devices". In: *IEEE transactions on visualization and computer graphics* 21.11 (2015), pp. 1241–1250.

[9] Thomas Whelan, Renato F Salas-Moreno, Ben Glocker, Andrew J Davison, and Stefan Leutenegger. "ElasticFusion: Real-Time Dense SLAM and Light Source Estimation". In: *IJRR* 35.14 (2016), pp. 1697–1716.

[10] Victor A Prisacariu, Olaf Kähler, Stuart Golodetz, Michael Sapienza, Tommaso Cavallari, Philip H S Torr, and David W Murray. "InfiniTAM v3: A Framework for Large-Scale 3D Reconstruction with Loop Closure". In: *arXiv preprint arXiv:1708.00783v1* (2017).

[11] Sotiris Malassiotis and Michael G Strintzis. "Robust real-time 3D head pose estimation from range data". In: *Pattern Recognition* 38.8 (2005), pp. 1153–1165.

[12] Simon Hadfield and Richard Bowden. "Generalised Pose Estimation Using Depth". In: *Trends and Topics in Computer Vision*. Ed. by Kiriakos N. Kutulakos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 312–325.

[13] Zhenyao Mo and Ulrich Neumann. "Real-time hand pose recognition using low-resolution depth images". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 2. IEEE. 2006, pp. 1499–1505.

[14] Brian Holt and Richard Bowden. “Static pose estimation from depth images using random regression forests and hough voting”. In: *VISAPP 2012-Proceedings of the International Conference on Computer Vision Theory and Applications*. Vol. 1. 2012, pp. 557–564.

[15] Tommaso Cavallari, Stuart Golodetz*, Nicholas A Lord*, Julien Valentin, Luigi Di Stefano, and Philip H S Torr. “On-the-Fly Adaptation of Regression Forests for Online Camera Relocalisation”. In: *CVPR*. 2017, pp. 4457–4466.

[16] Tommaso Cavallari*, Stuart Golodetz*, Nicholas A Lord*, Julien Valentin*, Victor A Prisacariu, Luigi Di Stefano, and Philip H S Torr. “Real-Time RGB-D Camera Pose Estimation in Novel Scenes using a Relocalisation Cascade”. In: *arXiv preprint arXiv:1810.12163* (2018).

[17] Siddhartha Chandra, Grigorios G. Chrysos, and Iasonas Kokkinos. “Surface Based Object Detection in RGBD Images”. In: *British Machine Vision Conference 2015*. 2015.

[18] Shuai Zheng, Victor Adrian Prisacariu, Melinos Averkiou, Ming-Ming Cheng, Niloy J. Mitra, Jamie Shotton, Philip H. S. Torr, and Carsten Rother. “Object Proposals Estimation in Depth Image Using Compact 3D Shape Manifolds”. In: *Pattern Recognition*. Ed. by Juergen Gall, Peter Gehler, and Bastian Leibe. Cham: Springer International Publishing, 2015, pp. 196–208.

[19] Christopher Lehnert, Inkyu Sa, Christopher McCool, Ben Upcroft, and Tristan Perez. “Sweet pepper pose detection and grasping for automated crop harvesting”. In: *ICRA*. 2016.

[20] Chen Chen, Roozbeh Jafari, and Nasser Kehtarnavaz. “UTD-MHAD: A multimodal dataset for human action recognition utilizing a depth camera and a wearable inertial sensor”. In: *Image Processing (ICIP), 2015 IEEE International Conference on*. 2015, pp. 168–172.

[21] Simon Hadfield and Richard Bowden. “Hollywood 3D: Recognizing Actions in 3D Natural Scenes”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2013.

[22] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi. “Edge Computing for Autonomous Driving: Opportunities and Challenges”. In: *Proceedings of the IEEE* (2019), pp. 1–20.

[23] Wiley J. Larson and James Richard Wertz. *Space Mission Analysis and Design*. 1992.

[24] Chen J Chiu, Wei Xian, and Cynthia F. Moss. “Flying in silence: Echolocating bats cease vocalizing to avoid sonar jamming.” In: *Proceedings of the National Academy of Sciences of the United States of America* 105 35 (2008), pp. 13116–21.

[25] Kourosh Khoshelham and Sander Oude Elberink. “Accuracy and Resolution of Kinect Depth Data for Indoor Mapping Applications”. In: *Sensors* 12.2 (2012), pp. 1437–1454.

[26] Oliver Wasenmüller and Didier Stricker. “Comparison of kinect v1 and v2 depth images in terms of accuracy and precision”. In: *Asian Conference on Computer Vision*. Springer. 2016, pp. 34–45.

[27] Craig L. Glennie and Derek D. Lichti. “Static Calibration and Analysis of the Velodyne HDL-64E S2 for High Accuracy Mobile Scanning”. In: *Remote Sensing* 2.6 (2010), pp. 1610–1624.

[28] John Frederick William Herschel. “On the parallax of the fixed stars”. In: *Philosophical Transactions of the Royal Society* 116 (1815), pp. 266–280.

[29] Clément Godard, Oisin Mac Aodha, and Gabriel J Brostow. “Unsupervised Monocular Depth Estimation with Left-Right Consistency”. In: *CVPR*. 2017.

[30] Fayao Liu, Chunhua Shen, Guosheng Lin, and Ian Reid. “Learning Depth from Single Monocular Images Using Deep Convolutional Neural Fields”. In: *T-PAMI* 38.10 (2016), pp. 2024–2039.

[31] Keisuke Tateno, Federico Tombari, Iro Laina, and Nassir Navab. “CNN-SLAM: Real-time dense monocular SLAM with learned depth prediction”. In: *CVPR*. 2017, pp. 6243–6252.

[32] D Marr and T Poggio. “Cooperative computation of stereo disparity.” In: *Science (New York, N.Y.)* 194.4262 (Oct. 1976), pp. 283–7.

[33] D Marr and T Poggio. “A computational theory of human stereo vision.” In: *Proceedings of the Royal Society of London. Series B, Containing papers of a Biological character. Royal Society (Great Britain)* 204.1156 (May 1979), pp. 301–28.

[34] Stephen T. Barnard and Martin A. Fischler. “Computational Stereo”. In: *ACM Comput. Surv.* 14.4 (Dec. 1982), pp. 553–572.

[35] D. Scharstein and R. Szeliski. “A taxonomy and evaluation of dense two-frame stereo correspondence algorithms”. In: *International journal of computer vision* 47.1 (2002), pp. 7–42.

[36] Myron Z Brown, Darius Burschka, and Gregory D Hager. “Advances in computational stereo”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 8 (2003), pp. 993–1008.

[37] Steven M Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. “A comparison and evaluation of multi-view stereo reconstruction algorithms”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Vol. June. 2006.

[38] Jure Zbontar and Yann LeCun. “Computing the Stereo Matching Cost With a Convolutional Neural Network”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[39] Wenjie Luo, Alexander G. Schwing, and Raquel Urtasun. “Efficient Deep Learning for Stereo Matching”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[40] Moritz Menze, Christian Heipke, and Andreas Geiger. “Object Scene Flow”. In: *ISPRS Journal of Photogrammetry and Remote Sensing (JPRS)* (2018).

[41] Moritz Menze, Christian Heipke, and Andreas Geiger. “Joint 3D Estimation of Vehicles and Scene Flow”. In: *ISPRS Annals* 2 (2015), p. 427.

[42] Andreas Geiger, Philip Lenz, and Raquel Urtasun. “Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite”. In: *CVPR*. 2012.

[43] John Woodfill and Brian Von Herzen. “Real-time stereo vision on the PARTS reconfigurable computer”. In: *FCCM*. 1997.

[44] Stefan K Gehrig and Clemens Rabe. “Real-Time Semi-Global Matching on the CPU”. In: *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*. IEEE. 2010, pp. 85–92.

[45] Christian Banz, Sebastian Hesselbarth, Holger Flatt, Holger Blume, and Peter Pirsch. "Real-time stereo vision system using semi-global matching disparity estimation: Architecture and FPGA-implementation". In: *SAMOS*. 2010.

[46] Christian Banz, Holger Blume, and Peter Pirsch. "Real-Time Semi-Global Matching Disparity Estimation on the GPU". In: *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*. IEEE. 2011, pp. 514–521.

[47] Gabriele Camellini, Mirko Felisa, Paolo Medici, Paolo Zani, Francesco Gregoretti, Claudio Passerone, and Roberto Passerone. "3DV-An embedded, dense stereovision-based depth mapping system". In: *Intelligent Vehicles Symposium Proceedings*. 2014.

[48] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *arXiv preprint arXiv:1906.02243* (2019).

[49] Andrew J. Davison. "FutureMapping: The Computational Structure of Spatial AI Systems." In: *arXiv preprint arXiv:1803.11288* (2018).

[50] Yannis Tsividis and Colin McAndrew. *Operation and Modeling of the MOS Transistor*. Vol. 2. Oxford university press Oxford, 1999.

[51] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[52] C. Mack. "The Multiple Lives of Moore's Law". In: *IEEE Spectrum* 52.4 (Apr. 2015), pp. 31–31.

[53] Ben Streetman. *Solid state electronic devices*. Boston: Pearson, 2015.

[54] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. "Leakage current: Moore's law meets static power". In: *Computer* 36.12 (Dec. 2003), pp. 68–75.

[55] O. S. Unsal and I. Koren. "System-level power-aware design techniques in real-time systems". In: *Proceedings of the IEEE* 91.7 (July 2003), pp. 1055–1069.

[56] J. Kao, S. Narendra, and A. Chandrakasan. "Subthreshold leakage modeling and reduction techniques [IC CAD tools]". In: *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.* Nov. 2002, pp. 141–148.

[57] David Patterson. *Computer organization and design : the hardware/software interface*. Waltham, MA: Morgan Kaufmann, 2012.

[58] Chandrakant D. Patel, Cullen E. Bash, and Abdlmonem H. Beitelmal. "Smart cooling of data centers". In: *2003 International Electronic Packaging Technical Conference and Exhibition, Volume 2*. 2003, pp. 129–137.

[59] Ronald Sass. *Embedded systems design with platform FPGAs : principles and practices*. Amsterdam Boston: Morgan Kaufmann, 2010.

[60] Tor Aamodt. *General-purpose graphics processor architectures*. San Rafael, California: Morgan & Claypool Publishers, 2018.

[61] Christian de Schryver, Ivan Shcherbakov, Frank Kienle, Norbert Wehn, Henning Marxen, Anton Kostiuk, and Ralf Korn. "An Energy Efficient FPGA Accelerator for Monte Carlo Option Pricing with the Heston Model". In: *2011 International Conference on Reconfigurable Computing and FPGAs*. 2011, pp. 468–474.

[62] David Barrie Thomas, Lee Howes, and Wayne Luk. "A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '09. Monterey, California, USA: ACM, 2009, pp. 63–72.

[63] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. "Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA?" In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 2012, pp. 232–239.

[64] Hanaa M. Hussain, Khaled Benkrid, Ahmet T. Erdogan, and Huseyin Seker. "Highly Parameterized K-means Clustering on FPGAs: Comparative Results with GPPs and GPUs". In: *2011 International Conference on Reconfigurable Computing and FPGAs*. 2011, pp. 475–480.

[65] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. "GPUs and the Future of Parallel Computing". In: *international symposium on microarchitecture* 31.5 (2011), pp. 7–17.

[66] Song Huang, Shucai Xiao, and Wu-chun Feng. "On the energy efficiency of graphics processing units for scientific computing". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. 2009, pp. 1–8.

[67] Simon McIntosh-Smith, Terry Wilson, Amaurys Ávila Ibarra, Jonathan Crisp, and Richard B. Sessions. "Benchmarking Energy Efficiency, Power Costs and Carbon Emissions on Heterogeneous Systems". In: *The Computer Journal* 55.2 (2012), pp. 192–205.

[68] Zachary K. Baker, Maya B. Gokhale, and Justin L. Tripp. "Matched Filter Computation on FPGA, Cell and GPU". In: *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. 2007, pp. 207–218.

[69] Sparsh Mittal and Jeffrey S. Vetter. "A Survey of Methods for Analyzing and Improving GPU Energy Efficiency". In: *ACM Computing Surveys* 47.2 (2015), p. 19.

[70] Kyle L. Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures". In: *Proceedings of the 9th conference on Computing Frontiers*. 2012, pp. 103–112.

[71] Juan M. Cebri'n, Ginés D. Guerrero, and José M. García. "Energy Efficiency Analysis of GPUs". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012, pp. 1014–1022.

[72] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David A. Patterson, John Shalf, and Katherine A. Yelick. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures". In: *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 2008, p. 4.

[73] Holger Lange, Florian Stock, Andreas Koch, and Dietmar Hildenbrand. "Acceleration and Energy Efficiency of a Geometric Algebra Computation using Reconfigurable Computers and GPUs". In: *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. 2009, pp. 255–258.

[74] Donald Bailey. *Design for embedded image processing on FPGAs*. Singapore: IEEE John Wiley & Sons (Asia, 2011.

[75] Dan Zou, Yong Dou, and Fei Xia. “Optimization schemes and performance evaluation of Smith–Waterman algorithm on CPU, GPU and FPGA”. In: *Concurrency and Computation: Practice and Experience* 24.14 (2012), pp. 1625–1644.

[76] S. Kestur, J. D. Davis, and O. Williams. “BLAS Comparison on FPGA, CPU and GPU”. In: *2010 IEEE Computer Society Annual Symposium on VLSI*. July 2010, pp. 288–293.

[77] Khaled Benkrid, Ali Akoglu, Cheng Ling, Yang Song, Ying Liu, and Xiang Tian. “High performance biological pairwise sequence alignment: FPGA versus GPU versus cell BE versus GPP”. In: *International Journal of Reconfigurable Computing* 2012 (2012), p. 7.

[78] Tsuyoshi Hamada, Khaled Benkrid, Keigo Nitadori, and Makoto Taiji. “A Comparative Study on ASIC, FPGAs, GPUs and General Purpose Processors in the $O(N^2)$ Gravitational N-body Simulation”. In: *2009 NASA/ESA Conference on Adaptive Hardware and Systems*. 2009, pp. 447–452.

[79] Daniel Llamocca, Cesar Carranza, and Marios S. Pattichis. “Separable FIR Filtering in FPGA and GPU Implementations: Energy, Performance, and Accuracy Considerations”. In: *2011 21st International Conference on Field Programmable Logic and Applications*. 2011, pp. 363–368.

[80] Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh, Ryan Kastner, and Scott B. Baden. “Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs”. In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. 2010, pp. 11–18.

[81] Matthias Birk, Matthias Balzer, Nicole V. Ruiter, and Jürgen Becker. “Comparison of processing performance and architectural efficiency metrics for FPGAs and GPUs in 3D Ultrasound Computer Tomography”. In: *2012 International Conference on Reconfigurable Computing and FPGAs*. 2012, pp. 1–7.

[82] I. Kuon and J. Rose. “Measuring the Gap Between FPGAs and ASICs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (Feb. 2007), pp. 203–215.

[83] Karl Pauwels, M. Tomasi, Javier Diaz Alonso, E. Ros, and M. M. Van Hulle. “A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features”. In: *IEEE Transactions on Computers* 61.7 (2012), pp. 999–1012.

[84] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. “A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications”. In: *FPGA*. 2012.

[85] Oscar Rahnama, Aleksej Makarov, and Philip Torr. “Real-time depth processing for embedded platforms”. In: *Real-Time Image and Video Processing 2017*. Vol. 10223. International Society for Optics and Photonics. 2017, 102230N.

[86] Andreas Geiger, Martin Roser, and Raquel Urtasun. “Efficient large-scale stereo matching”. In: *ACCV*. 2010.

[87] Daniel Tortei Tertei, Jonathan Piat, and Michel Devy. “FPGA design of EKF block accelerator for 3D visual SLAM”. In: *Computers & Electrical Engineering* 55 (2016), pp. 123–137.

[88] H Christopher Longuet-Higgins. “A computer algorithm for reconstructing a scene from two projections”. In: *Nature* 293.5828 (1981), p. 133.

[89] R. I. Hartley. “In defense of the eight-point algorithm”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19.6 (June 1997), pp. 580–593.

[90] Richard I. Hartley. “Theory and Practice of Projective Rectification”. In: *International Journal of Computer Vision* 35.2 (Nov. 1999), pp. 115–127.

[91] Richard Hartley. *Multiple view geometry in computer vision*. Cambridge, UK New York: Cambridge University Press, 2004.

[92] Simon Prince. *Computer vision : models, learning, and inference*. New York: Cambridge University Press, 2012.

[93] Juyang Weng, Paul Cohen, and Marc Herniou. “Camera calibration with distortion models and accuracy evaluation”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 10 (1992), pp. 965–980.

[94] Duane C Brown. “Decentering distortion of lenses”. In: *Photogrammetric Engineering and Remote Sensing* (1966).

[95] John G Fryer and Duane C Brown. “Lens distortion for close-range photogrammetry”. In: *Photogrammetric engineering and remote sensing* 52.1 (1986), pp. 51–58.

[96] H. Hu, C. Chen, B. Wu, X. Yang, Q. Zhu, and Y. Ding. “Texture-Aware Dense Image Matching Using Ternary Census Transform”. In: *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* (June 2016), pp. 59–66.

[97] D. Scharstein, R. Szeliski, and R. Zabih. “A taxonomy and evaluation of dense two-frame stereo correspondence algorithms”. In: *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*. 2001, pp. 131–140.

[98] Kristian Ambrosch and Wilfried Kubinger. “Accurate Hardware-based Stereo Vision”. In: *Comput. Vis. Image Underst.* 114.11 (Nov. 2010), pp. 1303–1316.

[99] T. Kanade and M. Okutomi. “A stereo matching algorithm with an adaptive window: theory and experiment”. In: *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. Apr. 1991, 1088–1095 vol.2.

[100] Y. Boykov, O. Veksler, and R. Zabih. “A variable window approach to early vision”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.12 (Dec. 1998), pp. 1283–1294.

[101] R.K. Gupta and S.Y. Cho. “A Correlation-Based Approach for Real-Time Stereo Matching”. In: *Advances in Visual Computing*. Ed. by Bebis G. et al. Vol. 6454. Lecture Notes in Computer Science (LNCS). Springer, Berlin, Heidelberg, 2010, pp. 129–138.

[102] Beau Tippetts, Dah Jye Lee, Kirt Lillywhite, and James Archibald. “Review of stereo vision algorithms and their suitability for resource-limited systems”. In: *Journal of Real-Time Image Processing* 11.1 (2016), pp. 5–25.

[103] R. Kalarot and J. Morris. “Comparison of FPGA and GPU implementations of real-time stereo vision”. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*. June 2010, pp. 9–15.

[104] S. Wong, S. Vassiliadis, and S. Cotofana. “A sum of absolute differences implementation in FPGA hardware”. In: *Proceedings. 28th Euromicro Conference*. 2002, pp. 183–188.

[105] A. Greeshma and G. Renu. “Object Tracking Using Filtering and Optimized Sum of Absolute Difference”. In: *International Journal of Engineering Research & Technology (IJERT)* 3.4 (Apr. 2014), pp. 2322–2325.

[106] W. Li and Z. Li. “Real-Time Stabilization Aboat Image with Big Target Based on SAD Algorithm”. In: *2012 Spring Congress on Engineering and Technology*. May 2012, pp. 1–4.

[107] Zoltán Kincses, László Orzó, Zoltán Nagy, György Mező, and Péter Szolgay. “High-Speed, SAD Based Wavefront Sensor Architecture Implementation on FPGA”. In: *Journal of Signal Processing Systems* 64.3 (2011), pp. 279–290.

[108] *Xilinx Zynq-7000 All Programmable SoC Overview, Product Specification, DS190 (v1.10)*. accessed on 9 February 2017. Xilinx. Sept. 2016.

[109] *Xilinx 7 Series FPGAs Configurable Logic Block, User Guide, UG474 (v1.8)*. accessed on 9 February 2017. Xilinx. Sept. 2016.

[110] D. P. Chaikalis, N. P. Sgouros, and D. E. Maroulis. “Real-time processing pipeline for 3D imaging applications”. In: *2009 16th International Conference on Digital Signal Processing*. July 2009, pp. 1–6.

[111] N. Isakova, S. Başak, and A. C. Sönmez. “FPGA design and implementation of a real-time stereo vision system”. In: *2012 International Symposium on Innovations in Intelligent Systems and Applications*. July 2012, pp. 1–5.

[112] L. Lin and W. Zhou. “Binocular Stereo Vision Synchronous Acquisition Technique Based on FPGA”. In: *2010 2nd International Workshop on Database Technology and Applications*. Nov. 2010, pp. 1–5.

[113] Tomasz Kryjak, Mateusz Komorkiewicz, and Marek Gorgon. “Real-time hardware–software embedded vision system for ITS smart camera implemented in Zynq SoC”. In: *Journal of Real-Time Image Processing* (2016), pp. 1–37.

[114] S. Ghosh and A. Saha. “Speed-area optimized FPGA implementation for Full Search Block Matching”. In: *25th International Conference on Computer Design*. Oct. 2007, pp. 13–18.

[115] Isaac TS Li, Warren Shum, and Kevin Truong. “160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)”. In: *BMC Bioinformatics* 8.1 (2007), p. 185.

[116] Kristian Ambrosch, Martin Humenberger, Wilfried Kubinger, and Andreas Steininger. “SAD-Based Stereo Matching Using FPGAs”. In: (2009), pp. 121–138.

[117] Carlos Cuadrado, Aitzol Zuloaga, Jose L Martin, Jesus Laizaro, and Jaime Jimenez. “Real-time stereo vision processing system in a FPGA”. In: *IECON*. 2006.

[118] Seunghun Jin, Jung Uk Cho, Xuan Dai Pham, Kyoung Mu Lee, Sung-Kee Park, Munsang Kim, and Jae Wook Jeon. “FPGA Design and Implementation of a Real-Time Stereo Vision System”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 20.1 (2010), pp. 15–26.

[119] Ratheesh Kalarot and John Morris. “Comparison of FPGA and GPU implementations of real-time stereo vision”. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*. 2010, pp. 9–15.

[120] John Iselin Woodfill, Gaile Gordon, Dave Jurasek, Terrance Brown, and Ron Buck. “The Tyzx DeepSea G2 Vision System, ATaskable, Embedded Stereo Camera”. In: *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW’06)*. 2006, pp. 126–126.

[121] Abdelelah Naoulou, J-L Boizard, Jean-Yves Fourniols, and Michel Devy. “An alternative to sequential architectures to improve the processing time of passive stereovision algorithms”. In: *Field Programmable Logic and Applications*. 2006.

[122] Stefania Perri, Daniela Colonna, Paolo Zicari, and Pasquale Corsonello. “SAD-based stereo matching circuit for FPGAs”. In: *Electronics, Circuits and Systems*. 2006.

[123] Lu Zhang, Ke Zhang, Tian Sheuan Chang, Gauthier Lafruit, Georgi Krasimirov Kuzmanov, and Diederik Verkest. “Real-time high-definition stereo matching on FPGA”. In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM. 2011, pp. 55–64.

[124] Kristian Ambrosch and Wilfried Kubinger. “Accurate hardware-based stereo vision”. In: *CVIU* (2010).

[125] H. Cui and N. Dahnoun. “Real-time stereo vision implementation on the Texas Instruments KeyStone II SoC”. In: *2016 IEEE International Conference on Imaging Systems and Techniques (IST)*. Oct. 2016, pp. 195–200.

[126] Oscar Rahnama, Duncan Frost, Ondrej Miksik, and Philip HS Torr. “Real-Time Dense Stereo Matching With ELAS on FPGA-Accelerated Embedded Devices”. In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 2008–2015.

[127] M Perez-Patricio, Abiel Aguilar-Gonzalez, M Arias-Estrada, Hector-Ricardo Hernandez-de Leon, Jorge-Luis Camas-Anzueto, and JA de Jesus Osuna-Coutiño. “An FPGA stereo matching unit based on fuzzy logic”. In: *Microprocessors and Microsystems* (2016).

[128] Giuseppe Cocorullo, Pasquale Corsonello, Fabio Frustaci, and Stefania Perri. “An efficient hardware-oriented stereo matching algorithm”. In: *Microprocessors and Microsystems* (2016).

[129] Heiko Hirschmuller. “Accurate and efficient stereo processing by semi-global matching and mutual information”. In: *CVPR*. 2005.

[130] Heiko Hirschmuller and Daniel Scharstein. “Evaluation of cost functions for stereo matching”. In: *CVPR*. 2007.

[131] Michael Drumheller and Tomaso Poggio. “On parallel stereo”. In: *ICRA*. 1986.

[132] Ali E Kayaalp and James L Eckman. “Near real-time stereo range detection using a pipeline architecture”. In: *IEEE Transactions on Systems, Man, and Cybernetics* (1990).

[133] T. Kanade, H. Kano, S. Kimura, A. Yoshida, and K. Oda. “Development of a video-rate stereo machine”. In: *IROS*. 1995.

[134] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. “A stereo machine for video-rate dense depth mapping and its new applications”. In: *CVPR*. 1996.

[135] Ramin Zabih and John Woodfill. “Non-parametric local transforms for computing visual correspondence”. In: *ECCV*. 1994.

[136] Masanori Hariyama, Yasuhiro Kobayashi, Haruka Sasaki, and Michitaka Kameyama. “FPGA implementation of a stereo matching processor based on window-parallel-and-pixel-parallel architecture”. In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* (2005).

[137] Christos Georgoulas, Leonidas Kotoulas, Georgios Ch Sirakoulis, Ioannis Andreadis, and Antonios Gasteratos. “Real-time disparity map computation module”. In: *Microprocessors and Microsystems* (2008).

[138] Daniel Scharstein and Richard Szeliski. “A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms”. In: *IJCV* (2002).

[139] Daniel Scharstein and Richard Szeliski. “High-accuracy stereo depth maps using structured light”. In: *CVPR*. 2003.

[140] Stefan K Gehrig, Felix Eberli, and Thomas Meyer. “A real-time low-power stereo vision engine using semi-global matching”. In: *ICVS*. 2009.

[141] Christos Georgoulas and Ioannis Andreadis. “A real-time fuzzy hardware structure for disparity map computation”. In: *Journal of Real-Time Image Processing* (2011).

[142] Martin Werner, Benno Stabernack, and Christian Riechert. “Hardware implementation of a full HD real-time disparity estimation algorithm”. In: *IEEE Transactions on Consumer Electronics* (2014).

[143] Wenqiang Wang, Jing Yan, Ningyi Xu, Yu Wang, and Feng-Hsiung Hsu. “Real-time high-quality stereo vision system in FPGA”. In: *IEEE Transactions on Circuits and Systems for Video Technology* (2015).

[144] Madaın Perez-Patricio and Abiel Aguilar-Gonzalez. “FPGA implementation of an efficient similarity-based adaptive window algorithm for real-time stereo matching”. In: *Journal of Real-Time Image Processing* (2015).

[145] Yanzhe Li, Kai Huang, and Luc Claesen. “SoC and FPGA oriented high-quality stereo vision system”. In: *FPL*. 2016.

[146] Minglun Gong and Yee-Hong Yang. “Near real-time reliable stereo matching using programmable graphics hardware”. In: *CVPR*. 2005.

[147] Jiangbo Lu, Gauthier Lafruit, and Francky Catthoor. “Fast variable center-biased windowing for high-speed stereo on programmable graphics hardware”. In: *ICIP*. 2007.

[148] Ines Ernst and Heiko Hirschmüller. “Mutual information based semi-global stereo matching on the GPU”. In: *ISVC*. 2008.

[149] Oscar Rahnama, Aleksej Makarov, and Philip H. S. Torr. “Real-Time Depth Processing for Embedded Platforms”. In: *Proceedings of the SPIE*. 2017.

[150] Daniel Hernandez-Juarez, Alejandro Chacon, Antonio Espinosa, David Vazquez, Juan Carlos Moure, and Antonio M Lopez. “Embedded real-time stereo estimation via Semi-Global Matching on the GPU”. In: *Procedia Computer Science* (2016).

[151] Oscar Rahnama, Tommaso Cavallari*, Stuart Golodetz*, Simon Walker, and Philip H S Torr. “R^{3}SGM: Real-time Raster-Respecting Semi-Global Matching for Power-Constrained Systems”. In: *FPT*. 2018.

[152] Leonardo De-Maeztu, Stefano Mattoccia, Arantxa Villanueva, and Rafael Cabeza. “Linear stereo matching”. In: *ICCV*. 2011, pp. 1708–1715.

[153] Asmaa Hosni, Michael Bleyer, and Margrit Gelautz. “Secrets of adaptive support weight techniques for local stereo matching”. In: *CVIU* 117.6 (2013), pp. 620–632.

[154] Asmaa Hosni, Christoph Rhemann, Michael Bleyer, Carsten Rother, and Margrit Gelautz. “Fast Cost-Volume Filtering for Visual Correspondence and Beyond”. In: *T-PAMI* 35.2 (2013), pp. 504–511.

[155] Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. “Stereo Matching Using Belief Propagation”. In: *T-PAMI* 25.7 (2003), pp. 787–800.

[156] Nikos Komodakis, Georgios Tziritas, and Nikos Paragios. “Fast, Approximately Optimal Solutions for Single and Dynamic MRFs”. In: *CVPR*. 2007.

[157] Frederic Besse, Carsten Rother, Andrew Fitzgibbon, and Jan Kautz. “PMBP: PatchMatch Belief Propagation for Correspondence Field Estimation”. In: *IJCV* 110.1 (2014), pp. 2–13.

[158] Qifeng Chen and Vladlen Koltun. “Fast MRF Optimization with Application to Depth Reconstruction”. In: *CVPR*. 2014, pp. 3914–3921.

[159] Heiko Hirschmuller. “Stereo processing by semiglobal matching and mutual information”. In: *T-PAMI* (2008).

[160] Korbinian Schmid and Heiko Hirschmüller. “Stereo Vision and IMU based Real-Time Ego-Motion and Depth Image Computation on a Handheld Device”. In: *ICRA*. 2013, pp. 4671–4678.

[161] Dominik Honegger, Helen Oleynikova, and Marc Pollefeys. “Real-time and Low Latency Embedded Computer Vision Hardware Based on a Combination of FPGA and Mobile CPU”. In: *IROS*. 2014, pp. 4930–4935.

[162] Yi Shan, Yuchen Hao, Wenqiang Wang, Yu Wang, Xu Chen, Huazhong Yang, and Wayne Luk. “Hardware Acceleration for an Accurate Stereo Vision System Using Mini-Census Adaptive Support Region”. In: *TECS* 13.4s (2014).

[163] Stefano Mattoccia and Matteo Poggi. “A passive RGBD sensor for accurate and real-time depth sensing self-contained into an FPGA”. In: *ICDSC*. 2015, pp. 146–151.

[164] Lu Zhang, Ke Zhang, Tian Sheuan Chang, Gauthier Lafruit, Georgi Krasimirov Kuzmanov, and Diederik Verkest. “Real-Time High-Definition Stereo Matching on FPGA”. In: *FPGA*. 2011, pp. 55–64.

[165] Christos Ttofis and Theocharis Theocharides. “Towards Accurate Hardware Stereo Correspondence: A Real-Time FPGA Implementation of a Segmentation-Based Adaptive Support Weight Algorithm”. In: *DATE*. 2012.

[166] Stefania Perri, Fabio Frustaci, Fanny Spagnolo, and Pasquale Corsonello. “Stereo vision architecture for heterogeneous systems-on-chip”. In: *JRTIP* (2018), pp. 1–23.

[167] Oscar Rahnama, Duncan Frost, Ondrej Miksik, and Philip H.S. Torr. “Real-Time Dense Stereo Matching With ELAS on FPGA-Accelerated Embedded Devices”. In: *RA-L* 3.3 (2018), pp. 2008–2015.

[168] Gabriele Facciolo, Carlo de Franchis, and Enric Meinhardt. “MGM: A Significantly More Global Matching for Stereovision”. In: *BMVC*. 2015.

[169] Moritz Menze and Andreas Geiger. “Object Scene Flow for Autonomous Vehicles”. In: *CVPR*. 2015.

[170] Daniel Scharstein, Heiko Hirschmüller, York Kitajima, Greg Krathwohl, Nera Nešić, X. Wang, and Porter Westling. “High-Resolution Stereo Datasets with Subpixel-Accurate Ground Truth”. In: *GCPR*. 2014, pp. 31–42.

[171] Amnon Drory, Carsten Haubold, Shai Avidan, and Fred A Hamprecht. “Semi-Global Matching: a principled derivation in terms of Message Passing”. In: *GCPR*. 2014, pp. 43–53.

[172] Jasmine Banks and Peter Corke. “Quantitative Evaluation of Matching Methods and Validity Measures for Stereo Vision”. In: *IJRR* 20.7 (2001), pp. 512–532.

[173] Heiko Hirschmuller and Daniel Scharstein. “Evaluation of stereo matching costs on images with radiometric differences”. In: *T-PAMI* 31.9 (2009).

[174] Heiko Hirschmüller. “Semi-Global Matching – Motivation, Developments and Applications”. In: *Photogrammetric Week 11* (2011), pp. 173–184.

[175] Ramin Zabih and John Woodfill. “Non-parametric Local Transforms for Computing Visual Correspondence”. In: *ECCV*. 1994.

[176] Andrey Kuzmin, Dmitry Mikushin, and Victor Lempitsky. “End-to-end Learning of Cost-Volume Aggregation for Real-time Dense Stereo”. In: *MLSP*. 2017, pp. 1–6.

[177] Oscar Rahnama, Tommaso Cavallari, Stuart Golodetz, Alessio Tonioni, Thomas Joy, Luigi Di Stefano, Simon Walker, and Philip H. S. Torr. “Real-Time Highly Accurate Dense Depth on a Power Budget Using an FPGA-CPU Hybrid SoC”. In: *IEEE Transactions on Circuits and Systems II-Express Briefs* 66.5 (2019), pp. 773–777.

[178] Stefania Perri, Fabio Frustaci, Fanny Spagnolo, and Pasquale Corsonello. “Design of Real-Time FPGA-based Embedded System for Stereo Vision”. In: *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE. 2018, pp. 1–5.

[179] Stefania Perri, Fabio Frustaci, Fanny Spagnolo, and Pasquale Corsonello. “Stereo vision architecture for heterogeneous systems-on-chip”. In: *Journal of Real-Time Image Processing* (2018), pp. 1–23.

[180] Christos Ttofis and Theocharis Theocharides. “High-quality real-time hardware stereo matching based on guided image filtering”. In: *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association. 2014, p. 356.

[181] M Dehnavi and M Eshghi. “FPGA based real-time on-road stereo vision system”. In: *Journal of Systems Architecture* 81 (2017), pp. 32–43.

[182] Daolu Zha, Xi Jin, and Tian Xiang. “A real-time global stereo-matching on FPGA”. In: *Microprocessors and Microsystems* 47 (2016), pp. 419–428.

[183] Beau Tippetts, Dah Jye Lee, Kirt Lillywhite, and James Archibald. “Review of Stereo Vision Algorithms and their Suitability for Resource-Limited Systems”. In: *Journal of Real-Time Image Processing* 11.1 (2016), pp. 5–25.

[184] Tobi Delbrück, Bernabe Linares-Barranco, Eugenio Culurciello, and Christoph Posch. “Activity-driven, event-based vision sensors”. In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 2010, pp. 2426–2429.

[185] Christoph Posch, Teresa Serrano-Gotarredona, Bernabé Linares-Barranco, and Tobi Delbrück. “Retinomorphic Event-Based Vision Sensors: Bioinspired Cameras With Spiking Output”. In: *Proceedings of the IEEE* 102.10 (2014), pp. 1470–1484.

[186] Donald G. Dansereau, Oscar Pizarro, and Stefan B. Williams. “Decoding, Calibration and Rectification for Lenselet-Based Plenoptic Cameras”. In: *2013 IEEE Conference on Computer Vision and Pattern Recognition*. 2013, pp. 1027–1034.

[187] Ren Ng, Marc Levoy, Mathieu Brédif, Gene Duval, Mark Horowitz, and Pat Hanrahan. “Light Field Photography with a Hand-held Plenoptic Camera”. In: *Computer Science Technical Report CSTR* 2.11 (2005).