Reducing FPGA Algorithm Area by Avoiding Redundant Computation

Brian Axelrod[†] Michel Laverne[‡]
Robotics Institute, Carnegie Mellon University

***Abstract*— We develop a new paradigm for designing fully streaming, area-efficient FPGA implementations of common building blocks for vision algorithm. By focusing on avoiding redundant computation we achieve a reduction of one to two orders of magnitude reduction in design area utilization as compared to previous implementations. We demonstrate that our design works in practice by building five 325 frames per second, high resolution Harris corner detection cores onto a single FPGA.**

***Index Terms*— FPGA, Vision, Accelerator, Harris Corner, Convolution, Non-Max Suppression**

I. INTRODUCTION

Field Robotics imposes hard design restrictions on robotics engineers. Constrained power and space limit computational resources. These limited computational resources must be shared between the localization, planning, control, and state estimation subsystems on the robot.

Carnegie Mellon University's entry in the DARPA Robotics Challenge, CHIMP (CMU Highly Intelligent Mobile Platform), was a great example of such restrictions. With limited on board processing power and degraded communication with the outside world, the visual odometry system was always fighting other robot subsystems for CPU time. This conflict led to design choices that ultimately decreased the system's tracking performance.

FPGAs (Field-Programmable Gate Arrays) are low–power, available in small, ruggedized, radiation-hardened form factors, and can process large amounts of data in parallel. This makes FPGAs an excellent candidate for offloading visual odometry's vision processing. Furthermore, with advances in the computing power of FPGAs, we can run the complete localization system on a single FPGA board that mixes programmable logic cores and low–level software running on an embedded ARM processor. However, algorithm design for the logic portion of the FPGAs differs significantly from that of CPUs. Processing the high–resolution video feed with the limited FPGA area proved infeasible with state–of–the–art algorithms.

To the best of our knowledge, no work has been done on the growth of FPGA resource utilization in vision and robotics applications. With n defined as the kernel radius, current state–of–the–art implementations of computations (such as convolutions) grow at the rate of $O(n^2)$ and quickly become infeasible for the kernel sizes required for high resolutions.

Convolutions are used in many computer vision algorithms and are often a computational bottleneck. Isotropic and Gaussian kernels are used to reduce noise and gradients kernels are used in Harris corners [1], Canny edge detection [2] and Histogram of Oriented Gradients [3] to compute moments of the image. Convolutions are also used at the lowest level of many deep-learning based image recognition systems often, with window sizes as large as $16px \times 16px$ [4].

We present a method to significantly reduce the area which common computer vision building blocks require on a FPGA by avoiding redundant computation. The method pipelines well achieving a throughput of one pixel per clock cycle without using a large area on the FPGA.

We examine redundant computation between adjacent image windows, and propose a method for caching and reusing the intermediate result in order to reduce the area–growth as a function of window radius. This enables us to fit many complex operations onto the FPGA fabric and process higher–resolution images requiring larger radius windows.

We demonstrate our FPGA algorithm design-paradigm on two building blocks of computer vision—convolutions and Non-Max Suppression. Under the assumption of some structure, we improve convolution area growth with respect to kernel radius from $O(n^2)$ to $O(n)$ and Non-Max Suppression (NMS) from $O(n^2)$ to $O(\log n)$.

Finally, we apply our efficient convolutions and NMS to experimentally demonstrate a Harris corner detector implementation with extensive filtering. While it takes little area on the FPGA, it can process 1241×376 pixel images at over 300 frames per second.

II. BACKGROUND AND RELATED WORK

A. Introduction to FPGA Algorithm Design Constraints

The hard-logic imposed restrictions on FPGA algorithms force different design methodologies than those used to design for CPUs and GPUs (Graphical Processing Units). First, FPGAs operate at clock speeds at least an order of magnitude less than traditional CPU-based systems. Even using a state of the art FPGA architecture [5], the implementation we detail later in this paper operates at a frequency of 166MHz.

Although FPGAs operate at much lower clock frequencies than traditional processors, they can bring much more raw computational power to bear through parallelization and pipelining.

[†]NREC intern, now at MIT
baxelrod@csail.mit.edu
[‡]NREC Commercialization Specialist
mlaverne@nrec.ri.cmu.edu

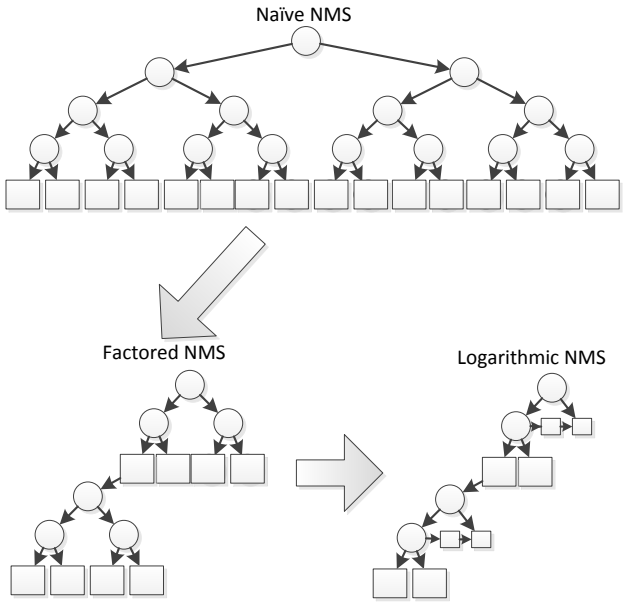

Fig. 1: A comparison of the logic-complexity of different FPGA NMS implementations. Squares represent data storage and circles represent comparisons. Factored NMS and Logarithmic NMS are detailed in this paper.

One way in which FPGAs allow parallelization is simply by making n side-by-side copies of the same logic. These copies of the logic can be used in parallel, increasing the amount of raw computation by a factor of n. This, however, also increases the FPGA resource use by a factor of n.

Most major speedups achieved when porting to FPGAs come from pipelining. In pipelining, a set of sequential operations are all executed in parallel on different pieces of data. Consider the case where we have some logic to decide what number to multiply each pixel. Our logic might be represented as a tree. A CPU would process the logic as illustrated in Figure 2, executing each step per cycle. The CPU only starts processing pixel 2 at time 3, after it has finished processing pixel 1.

Furthermore, standard analysis of the algorithm takes advantage of shorter paths in the logic to reduce the amortized cost.

Pipelining aims to utilize every computational resource during every cycle. At every cycle, a new value is fetched from memory and each previous value advances down the tree. Thus, after the pipeline is filled, the FPGA produces a new output during every clock cycle. At best, this speeds processing by a factor of three in this example, demonstrated in figure 3.

This allows FPGAs to achieve massive speedups on complicated operations with deep pipelines—they are always executing every step simultaneously. Amortized algorithms do not port well to FPGAs, however. Notice how, in Figure 3, pixel 1 must be propagated for the total length of the pipeline. The FPGA is not able to finish processing it early

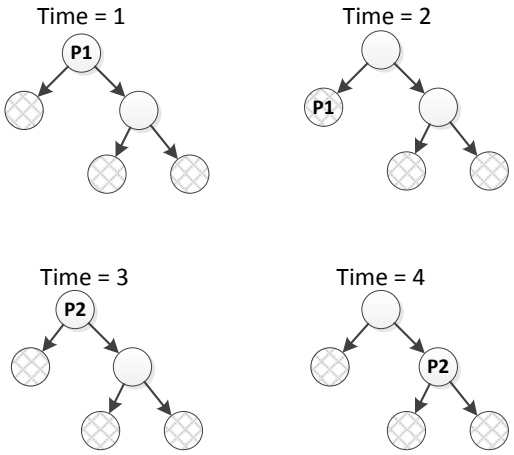

Fig. 2: A CPU executing logic. It finishes processing the first pixel on the second cycle because it can take advantage of the shorter logic path associated with the particular pixel value. After it is done processing the first pixel, it starts processing the second pixel.

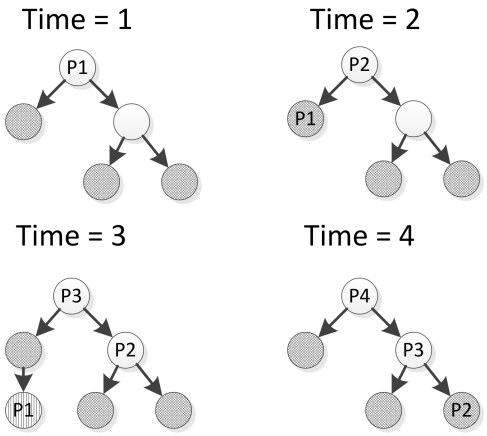

Fig. 3: An FPGA executing logic. The first pixel must go through the entire pipeline even though processing is completed at time 2.

because it is already publishing one output per clock cycle. Also note that even though the FPGA logic executed one side of a branch, the logic devoted to both sides of the branch is still there. One side sits idle. This leads to the issue of an appropriate cost metric for designing FPGA algorithms. While on a CPU the algorithm runtime is an appropriate cost metric, all properly pipelined FPGA algorithms are going to produce one output per clock cycle, regardless of how difficult the output was to compute. Instead of runtime, FPGA designers must minimize the amount of logic in their designs, i.e., the area of their design. Traversing a logical tree makes sense on the CPU where execution can skip large chunks of the logic. But on FPGAs, the logic is programmed into the fabric of the FPGA and remains idle when not used.

Furthermore, it limits the complexity of algorithms that can fit in the remaining space.

Dependence on neighboring data requires more consideration in a pipelined setting than in a parallelized setting. If a step of the pipeline requires data which has not yet been computed or is not yet available, the entire pipeline will stall until that data becomes available. If the pipeline was waiting on data generated in earlier part of the pipeline, it will stall indefinitely.

B. Related Work

There has been significant previous work in accelerating computer vision algorithms on FPGAs [6], [7], [8], [9], [10]. This work falls into two categories —implementing full computer vision algorithms (such as feature detectors), and implementing the basic building blocks of computer vision algorithms (such as convolution).

Most full implementations of computer vision algorithms, such as SURF, are concerned exclusively with throughput [6], [7]. Pauwels et al. compared an FPGA-accelerated optical flow system to a GPU accelerated system and demonstrate a large speedup [8]. These works, however, are rarely concerned with FPGA resource utilization as long as their design fits and operates on their FPGA of choice. Additionally, no work has been done on accelerating optimal features for modern optical flow/visual odometry systems, such as Badino's Multi-Frame Integration (MFI), that use Harris corners [11]. MFI is a particularly promising candidate for running on embedded systems because it provides state–of–the–art performance wth a less expensive optimization step than traditional bundle adjustment-based algorithms [11].

Work concerning the building blocks of computer vision algorithms such as convolution [12], [13] also tends to focus on throughput and comparing the performance of FPGA implementations and GPU implementations of the same operation. While they demonstrate a massive speedups with respect to GPU implementations, their convolution implementations are very inefficient in terms of area, with Cope et al.'s implementation of an 11x11 convolution taking up 99% of the area of their FPGA [13]. This makes implementing detectors like the Harris corner detector difficult, as they require many convolutions.

While the work on FPGA convolution is extensive, work on many other useful computer vision building blocks can be lacking. For example, while NMS has been extensively studied for the CPU, little work has been done with NMS on FPGAs. The state–of–the–art NMS algorithms rely on amortization and as such do not port well to FPGAs [14].

III. Method

A. General

The basic premise that we use to reduce FPGA area is that adjacent windows have redundant computation. Since pipelining forces us to apply the same logic to every pixel, we must improve the complexity of the entire logic pathway that can be executed. This requires splitting the problem into distinct steps, and in later steps using a neighborhood of values from the previous step. Avoiding redundant computation reduces the amount of per-pixel logic and thus use less FPGA resources. Splitting the process into distinct, sequential steps allows to easily identify inter–step dependencies and prevent pipeline stalls. The Kernel radius is defined as n for the rest of the discussion.

B. Convolution

The first example to which we apply our paradigm is convolution. The standard way of performing a convolution on an FPGA is by taking a window in the input image, multiplying each element by the corresponding kernel (resulting in $O(n^2)$ multiplications), and then feeding the resulting output into an adder tree ($O(n^2)$ additions) [12], [13].

However, this general convolution does not take advantage of structure in the kernel, which can result in redundant computation between adjacent windows. In vision applications, we can use structure in the kernel to identify redundant computation.

We identify structure in the kernel by doing a low-rank approximation. Consider the Singular Value Decomposition of the kernel $K = U\Sigma V$. Take the approximation of K, $\hat{K} = \sum_{i=1}^{k} u_i \sigma_i v_i$ with u_i and v_i the ith column and row vectors of U and V respectively. The values of Σ indicate how strongly your approximation is improved by including the particular vector pair. When considering the convolution, we can see how this allows us to break it up:

$$\begin{aligned} G * K \quad \approx G * \left(\sum_{i=1}^{k} u_i \sigma_i v_i \right) \quad &= \sum_{i=1}^{k} (G * u_i \sigma_i v_i) \\ &= \sum_{i=1}^{k} ((G * u_i) * (\sigma_i v_i)) \end{aligned}$$

Each vector pair can be split into a vertical and horizontal convolution. For each pair, we create logic for convolving by one vector that feeds a window-accumulator. This is turn feeds the convolution by the other vector. The intermediate windowing is key and allows us to remain perfectly pipelined. We create parallel logic for each pair of vectors and sum the results with an adder tree. For an illustration of the process see Figure 4.

In practice, convolution kernel often factor as a perfect outer product of two vectors, i.e., kernels are often separable. This means that we can often let $k = 1$ and forgo the adder tree after the individual convolutions.

Gaussian, isotropic average, and gradient kernels are all separable and can thus be perfectly factorized into a outer product of vectors. Furthermore, any matrix or 2d function can be approximated as a sum of outer products.

The factorization is a technique that has been used on CPU and GPU algorithms before, as it reduces the computational complexity. However, previous work fails to adapt the strategy to an effective pipelined algorithm [13], [12]. We know then for an image G, if $HV = K$ then $G * K = (G * H) * V$. We now can simply apply the same convolution methods

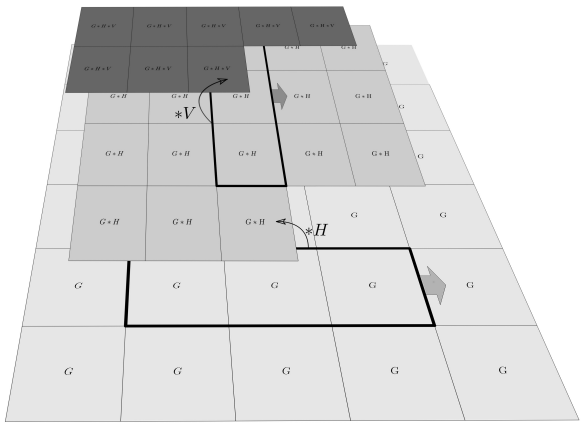

Fig. 4: Factorized Convolution process. From the input image G, the convolution by the horizontal vector $*H$ is applied sequentially. The output of the horizontal vector convolution is then used for the vertical vector convolution $*V$ to provide the output $G*H*V$. Note how the last processing step reuses the results from the previous convolution.

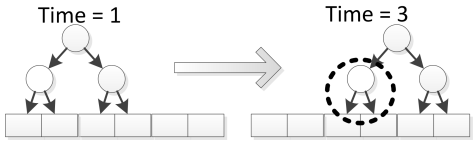

Fig. 5: A Non-Max Suppression Comparison Tree as it advances through two iterations. Note that the comparison circled was computed two cycles ago.

developed previously, but on $n \times 1$ sized vectors instead of a single $n \times n$ matrix. If the kernel is not separable, we can approximate the convolution by doing multiple convolutions via the eigenvectors of K, and then summing them.

In terms of resources, this approach applies $2k$, $1 \times n$ convolutions and uses only $O(n)$ resources. This represents a significant improvement over the naïve approach of simply multiplying every element by its corresponding value in the kernel and then summing them, which uses $O(n^2)$ resources. The final adder tree in our approach requires $O(k)$, resources which is insignificant in practice in comparison to $O(n)$.

C. Non-Max Suppression (NMS)

A similar idea can be applied to NMS as well. In order to be a maxima in a $n \times n$ window, a pixel must be the maxima of its own row, and larger than the maxima of the neighboring rows. Using this idea, we can apply a similar technique to the factored convolution mentioned above. The difference is that we use comparison trees instead of adder trees with multiplication at the leaves.

This reduces resource utilization to $O(n)$, compared to the naïve comparison tree which requires $O(n^2)$. However, adjacent row and column comparisons still involve large amounts of redundant computation because we are always comparing the same pixels. this can can be seen in Figure 5.

Instead of recomputing the comparison each time, we can store the result in a FIFO (First In First Out) queue. Note that the FIFO queue for a level i has to be of length $\frac{n}{2^i}$, where n is the length of the segment considered, and $i = 1$ corresponds to the top level. Thus the total sum of FIFO lengths is $O(n)$, which is the same as the original window. We then remove all but the leading edge of the comparison tree, leaving only the $O(\log n)$ comparisons as can be see in Figure 2.

IV. Results

We implemented all examples in Vivado and Vivado HLS for a Xilinx Zynq Z-7100 FPGA. Vivado HLS is a tool that generates Verilog and VHDL code from C++ code. Computational structures such as adder trees were generated with C++ template meta–programming.

Here we examine the utilization of different resources on the FPGA of the aforementioned algorithms, and compare them to the Xilinx reference implementations which are similar to that of [12], [13]. We provide a quick big-O analysis for each reference implementation.

A. Convolution Results

The standard convolution multiplies each element in the window, then adds them. This results in n^2 multiplications and n^2 summations, and thus uses $O(n^2)$ resources. The method we described above requires doing n multiplications and additions in the first stage and n in the second, requiring $2n$ multipliers and adders. It thus uses $O(n)$ resources. The below chart lists the predicted resource usage after HLS synthesis. Optimized utilization is usually about $\frac{2}{3}$ of the predicted values. The ultimate utilization values were not used because synthesis and implementation were computationally infeasible for the larger window sizes of the naïve algorithm.

Operation	BRAM	DSP	FF	LUT
Optimal 5x5	24	46	5,791	5,791
Reference 5x5	20	77	18,255	44,305
Improvement	0.8x	1.7x	3.2x	7.7x
Optimal 7x7	32	66	8,311	12,916
Reference 7x7	28	149	34,341	82,118
Improvement	0.9x	2.3x	4.1x	6.4x
Optimal 11x11	48	106	13,833	19,928
Reference 11x11	44	365	79,815	188,935
Improvement	0.9x	3.4x	5.8x	9.5x

Our algorithm demonstrates the expected linear growth, outperforming the naïve 11×11 convolution by a 3.4 times improvement in DSPs utilization, 5.8 time improvement in Flip Flops and 9.5 time improvement in Look-Up-Tables. The increase in Block RAM (BRAM) usage is to be expected

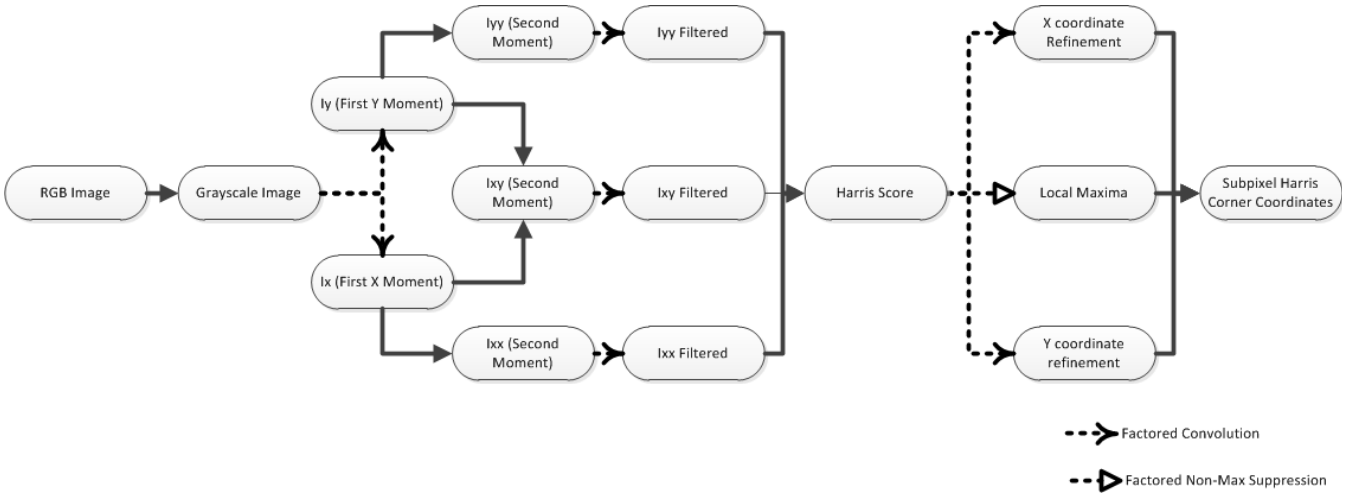

Fig. 6: Flowchart of the FPGA Harris corner implementation

because we partition our memory into more pieces, and thus cannot use BRAM as efficiently.

B. Non-Max Suppression Results

The naïve NMS method creates a max-tree over the entire window. This tree again has n^2 nodes and thus requires $O(n^2)$ resources. Our optimal comparison tree requires only $O(\log n)$ comparisons and thus $O(\log n)$ computational resources. Because our chosen window sizes are relatively small, the decrease from $O(n)$ to $O(\log n)$ did not yield significant improvement in design area.

Values predicted after HLS synthesis are shown below:

Operation	BRAM	DSP	FF	LUT
Linear 5x5	29	0	2,814	4,570
Reference 5x5	29	0	2,466	6,567
Improvement	1x	1x	0.9x	1.4x
Linear 7x7	37	0	4,010	6,986
Reference 7x7	37	0	4,790	12,498
Improvement	1x	1x	1.2x	1.8x
Linear 11x11	53	0	6,669	10,180
Reference 11x11	53	0	11,800	30,184
Improvement	1x	1x	1.8x	3.0x

For the NMS the gains are much less pronounced in the smaller window sizes. This is due to floating point comparisons being significantly cheaper operations than floating point addition and multiplication.

C. Harris Corner Detector Results

Lowering the resource utilization of the basic building blocks of computer vision algorithms (such as convolution and NMS) allows complex vision processing to be accelerated even on modest FPGAs.

Consider the following Harris corner detector detailed in [15]. It first convolves the image by a 7×7 kernel to get the x moment, and repeats this convolution for the y moment. It then uses the x and y moments to compute the second moments of the image. The three second moments are each smoothed with a 5×5 kernel, and the filtered moments are used to compute a Harris corner score. Then, through simultaneous NMS and quadratic interpolation, local maxima in a 5×5 window are identified and their coordinates are estimated to sub–pixel precision. Quadratic interpolation is done using two 3×3 kernels to identify best-fit coefficients of a paraboloid on the neighboring Harris Score. A flowchart illustrating the method is presented in Figure 6.

Since each element of the algorithm is perfectly pipelined we can do the entire computation efficiently on the FPGA.

The complete implementation runs at 325 frames per second for 1241×376 pixel images on a Xilinx Zynq Z-7100 using the following predicted resources:

	BRAM	DSP	FF	LUT
Harris Core	213	346	57,462	83,141
Z-7100 FPGA	755	2,020	554,800	277,400

V. Conclusion

We presented a novel way of designing FPGA algorithms that exploits redundant computation in adjacent windows to reduce FPGA resource usage. We also presented a big-O analysis of FPGA area as a function of window size—the first time this has been done for FPGA-based vision algorithms to the knowledge of the authors.

Smaller, more efficient convolution and NMS allow the implementation of many complex algorithms such as the Harris Corners detector with larger window sizes than was previously possible. The ability to run these algorithms efficiently on an FPGA opens up the ability to use advanced, real-time, computer vision processing to many embedded and ruggedized applications. Furthermore the smaller area corresponds to lower cost and power through the use of smaller FPGAs. This also leads to the possibility of including efficient, dedicated logic in ASIC without sacrificing die area or power consumption.

ACKNOWLEDGMENT

The work presented in this paper is based upon research support by the Intel Science and Technology Center for Embedded Computing at Carnegie Mellon University, Pittsburgh, PA.

The authors would also like to acknowledge Dane Bennington, Michael George and Jean-Philippe Tardif for their valuable contributions.

REFERENCES

[1] C. Harris and M. Stephens, "A combined corner and edge detector." in *Alvey Vision Conference*, vol. 15. Manchester, UK, 1988, p. 50.

[2] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 6, pp. 679–698, 1986.

[3] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 1. IEEE, 2005, pp. 886–893.

[4] B. A. Olshausen *et al.*, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, no. 6583, pp. 607–609, 1996.

[5] "Xilinx 7 series product brief," http://www.xilinx.com/publications/prod_mktg/7-Series-Product-Brief.pdf, accessed: 2014-10-01.

[6] J. Svab, T. Krajnik, J.Faigl, and L. Preucil, "FPGA-based speeded up robust features," in *2009 IEEE International Conference on Technologies for Practical Robot Applications 2009*, November 2009.

[7] N. Battezzati, S. Colazzo, M. Maffione, and L. Senepa, "SURFalgorithm in FPGA: A novel architecture for high demanding industrial applications," in *Design, Automation Test in Europe Conference Exhibition*, March 2012, pp. 161–162.

[8] K. Pauwels, M. Tomasi, J. Diaz Alonso, E. Ros, and M. Van Hulle, "A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features," *Computers, IEEE Transactions on*, vol. 61, no. 7, pp. 999–1012, July 2012.

[9] P. Athanas and A. Abbott, "Real-time image processing on a custom computing platform," *Computer*, vol. 28, no. 2, pp. 16–25, Feb 1995.

[10] B. Draper, J. Beveridge, A. P. W. Bohm, C. Ross, and M. Chawathe, "Accelerated image processing on FPGAs," *Image Processing, IEEE Transactions on*, vol. 12, no. 12, pp. 1543–1551, Dec 2003.

[11] H. Badino, A. Yamamoto, and T. Kanade, "Visual odometry by multi-frame feature integration," in *International Workshop on Computer Vision for Autonomous Driving @ ICCV*, December 2013.

[12] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of FPGA, GPU and CPU in image processing," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Aug 2009, pp. 126–131.

[13] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt, "Have GPUs made FPGAs redundant in the field of video processing?" in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, Dec 2005, pp. 111–118.

[14] A. Neubeck and L. V. Gool, "Efficient non-maximum suppression," in *ICPR 2006*, August 2006, in press.

[15] J. Shi and C. Tomasi, "Good features to track," in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR '94., 1994 IEEE Computer Society Conference on*, Jun 1994, pp. 593–600.