# Real-time FPGA-implementation for blue-sky Detection

Nhut Thanh Quach
LogicaCMG Nederland B.V.
Computer Engineering lab.
Delft Univ. of Technology
tquach@ce.et.tudelft.nl

Bahman Zafarifar
LogicaCMG Nederland B.V.
Philips Consumer Electronics
Eindhoven Univ. of Technol.
B.Zafarifar@tue.nl

Georgi N. Gaydadjiev
Computer Engineering lab.
Delft Univ. of Technology
The Netherlands
georgi@ce.et.tudelft.nl

## Abstract

*Currently, television sets with flat plasma and LCD screens with improved resolutions and better color quality are emerging. To fully utilize their capabilities, lower resolution Standard Definition video material is enhanced. During such process, existing noise can become clearly visible, or additional artifacts may be introduced. These impairments are usually better visible in smooth image areas such as sky regions, motivating the development of special techniques for their removal. In this paper, we introduce a hardware accelerator for an existing pixel-accurate and spatially-consistent sky-detection algorithm. We describe the algorithmic and architectural design considerations of a resource-efficient real-time system, targeting an FPGA platform. Our results show that it is feasible to implement a simplified algorithm version by using only 5,756 logic- and 23,687 memory elements of the targeted device. A demonstrator setup using real-time camera signal, proves that images of up to 640x480 at a frame rate of 30 fps can be processed. Furthermore, according to our estimations, images with pixel rates up to 142 MHz, e.g. High Definition TV, can be processed by the proposed system.*

## 1 Introduction

Algorithms to improve the picture quality of Standard Definition video broadcast streams are widely applied in the latest high resolution LCD and plasma TV sets. Besides their advantages, certain shortcomings of such algorithms can be detected due to conflicting requirements. For example, the sharpness enhancement being suitable for textured areas could affect the quality of uniformly colored (smooth) areas. On the other hand, the noise reduction techniques applicable to uniformly colored fields is not suitable for textured image sections. In such cases compromised settings are normally used for the interfering processing methods, often leading to suboptimal image quality, or sometimes even to introduction of artifacts. As the artifacts are more objectionable on large uniformly colored areas, the detection of commonly seen smooth surfaces, such as sky areas, is an emerging problem. Various blue-sky detection algorithms are proposed in the literature [3, 4, 5, 6, 7]. Even though these algorithms can be functionally correct, no real-time implementation has been reported. The real-time aspects are important as the desired implementation needs to be able to process video material in real-time and be both spatially and temporally consistent. Spatial consistency means that adjacent sky areas are assigned similar sky probabilities, while temporal consistency implies that the segmentation results should not change abruptly over time, when the actual image sequence does not cause this. Therefore, the implementation of the sky detector needs to be pixel and frame accurate, meaning that for all pixels of each frame an output should be calculated.

The main contributions of this paper are:

- Proposing a simplified version of a sky-detection algorithm suitable for real-time hardware implementation;
- Designing a hardware architecture of the above algorithm for real-time operation of SD video material;
- Implementing and performance analysis of the proposed architecture.

To the authors best knowledge, this is the first real-time implementation of a blue-sky detection algorithm reported in the literature.

The rest of this paper is organized as follows. Section 2 briefly explains a suitable blue-sky detection algorithm and describes the algorithm adjustments needed for a real-time HW implementation. In Section 3, we present our architectural design including the implementation results. Section 4 describes the experimental framework and discusses the results obtained from the experiments. Finally, Section 5 concludes the paper.

1-4244-1027-4/07/$25.00 ©2007 IEEE

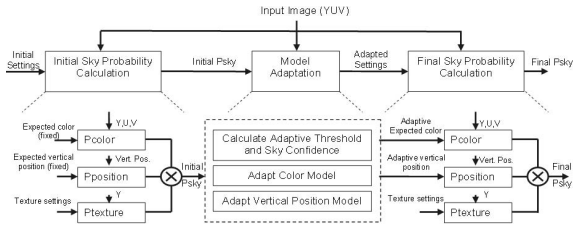


**Figure 1. Blockdiagram of the sky-detection algorithm**

## 2 The Sky-detection algorithm

The sky-detection algorithm proposed in [6, 7] is chosen for its spatial and temporal consistency, and the suitability of the employed techniques for a real-time embedded implementation. The algorithm is based on the following observations: a) blue sky regions are more likely to appear at the top of the image; b) they cover a certain part of the color space, and c) have smooth texture. In addition, the limited horizontal and vertical gradients (in the luminance channel) of sky regions are also considered. The algorithm contains the following three stages, as depicted in Fig. 1 (from left to right): *Stage 1*: Initial sky probability calculation; *Stage 2*: Model adaptation; *Stage 3*: Final sky probability calculation.

In the initial sky probability calculation stage, a sky-probability map is produced based on the color, vertical position, texture and gradient of the image pixels, using fixed, predefined settings. These settings are defined such that all desired sky appearances can be captured [4]. The model-adaptation stage adapts the fixed settings of the first stage to the properties of the processed image. As such, the settings for the vertical-position are adapted to the vertical position of pixels with high initial sky probability. Similarly, the settings for the color probability are adjusted to the the color values of pixels with high initial sky probability. The final sky probability calculation stage uses the adapted settings to create a pixel-accurate final sky-probability map based on a combination of the color, vertical position and the texture of the image pixels.

### 2.1 Algorithm adaptations

For the implementation of the selected algorithm we consider a hardware platform with limited resources for both the available logic and on-chip memory. We selected Altera EP1S10 Stratix FPGA as our hardware platform. This device incorporates 10,570 logic elements and 920 kb of on-chip memory [1]. In order to meet these self-imposed memory and resource constraints, we modified the original algorithm [6, 7] as follows. The initial multi-resolution texture analysis is simplified to a single-resolution analysis. The 2-dimensional spatially-adaptive color model is simplified to an improved version of the color model proposed in [4]. All the above modifications will lead to some false rejection of parts of the sky area. For our simplified proof of concept, these are not relevant. More details about the different simplifications is presented in the next sections.

#### 2.1.1 Initial multi-resolution texture analysis

The original algorithm utilizes a multi-resolution texture analysis in *Stage 1*. The texture analysis assigns low probabilities to parts of the image containing high luminance variations. This to exclude the textured areas from the initial sky probability. In this multi-resolution analysis three down-scaled versions (with factors of 2) of the luminance channel are analyzed using a fixed $5\times5$ pixels window-size. The results are combined in the lowest resolution, using the minimum operator.

In our implementation we include the texture analysis, but we evade the multi-resolution analysis. The latter is not implemented, because a serial implementation of this analysis would exceed the amount of on-chip memory available. This is due to the fact that the input image and the intermediate results require temporal storage. On the other hand, a parallel implementation of this analysis is envisioned to surpass the available logic resources of the targeted device. We implemented the texture analysis, along with the additional analysis in the initial sky probability calculation stage and the model-adaptation stage on a down-scaled version of the original image (down-scale factor 4). This is to expand the effective size of the texture analysis window, and to save on the amount of computations. The evation of the multi-resolution analysis leads to more false positives of lower frequency texture.

#### 2.1.2 2-dimensional spatially-adaptive color model

The 2-dimensional (2-d) spatially-adaptive color model of the original algorithm deals with the wide range of sky color values within one frame. For this purpose, each signal component is modeled by a spatially-varying 2-d function, that is fitted to a selected set of pixels with high initial sky probability.

In our simplified version, we have evaded the implementation of the 2-d spatially-adaptive color model. We implement a less complex color model proposed in [4]. This color model can deal with wide color ranges *between* different fields, but in cases when the input image contains a wide color range *within* the same field, a part of the sky will be rejected. We compensate for such false rejection of the sky areas by including a disparity analysis of the sky color (pixels with high initial sky probability). We examine

the color differences within these areas and when the color range is large, we increase the variance of the color model. Also the mean color of the color model is moved towards a more saturated blue color, to make the possible rejection of sky regions take place in the lower part of the image. This prevents the possible rejections to severely affect the post-processed image in application such as color enhancement. The color model with these compensations are still not as good as the adaptive color model.

#### 2.1.3 Final texture analysis

The original algorithm [6, 7] consists of an optional texture analysis in the final sky probability calculation. Using a large texture window in the final sky probability leads to a decreased sky probability around objects adjacent to sky areas, which subsequently can cause artifacts in the post-processed image. For example, in color enhancement, the color of sky areas in the post-processed image is saturated, except for a halo around objects next to sky areas which retains the original color. A moderate amount of this decrease of sky probability around the objects may be useful for some applications, such as noise reduction, for retaining the sharpness of the edges. In our version of the algorithm, the final sky probability calculation is only based on the color and vertical position of the pixels.

## 3 Architecture

This section describes the considerations that have lead to the architectural design of the adjusted algorithm. Our input image is in the YUV color-space and the image is interlaced scanned. In this scanning mode, each frame consists of two fields, containing the even and odd lines.

In Section 3.1, the memory design considerations are stated. The following sections describe the considerations in the three stages of the algorithm.

### 3.1 Design considerations

Consumer video appliances, such as TVs that are the main targets for our sky-detection system imply restricted amount of on-chip memory. In our case, the proposed sky-detection system is meant to be combined with an already existing video chain, to enable content-based locally-adaptive processing of sky areas. The permitted latency depends on the architecture of the video processing system, and where the sky-detection algorithm is placed. In a system which already includes an input/output latency, if the sky-detection system is placed at the appropriate position, a field or frame delay could be permissible. Obviously, the permitted delay in a memoryless architecture, would not be in terms of fields or frames, but based on the available data-buffering resources would be limited to a number of pixel- or line periods.

On the other hand, the algorithm requires the entire image data for calculating some of the parameters. For example, two threshold levels are calculated by examining the initial sky probability of the entire image, meaning that they cannot be calculated before all image lines are received. Similarly, the model-adaptation stage can start only after the two threshold levels are calculated, and the final sky probability can be computed only after the color model and the vertical-position models are updated. These data dependencies prescribe the sequence of the computation, and will therefore require the image data and the intermediate results (such as the initial sky probability) to be saved in relatively large amount of local memory. Besides, such sequential method also leads to a large input-output latency.

Special measures are needed to overcome the aforementioned problems. In the first place, our implementation uses a pixel-synchronous approach, meaning that instead of first storing the incoming image data in a field memory, and starting the computations only after the complete image is received, we temporarily store the incoming image lines in a small number of line-memories and start the computations as soon as the necessary image data are received (thus synchronous to the input pixel clock). In the second place, where parameters are needed before computations can start, we use the parameters that were computed in the previous field. To be more concrete, we take the threshold levels calculated in the previous field for model adaptation, and we use the color and vertical-position models of the previous field for final sky probability calculation. In this way, we can perform all computations (initial sky probability, threshold calculation, model adaptation and final sky probability) in parallel, albeit using one or two field old data. This approach not only improves the memory requirements drastically (no field-memory is required), but also decreases the input-output latency from order of fields to order of pixels.

It has to be noted that using the information from previous fields could lead to undesired side effects in case of significant temporal changes in the color values of the image, e.g. due to object or camera motion, or rapid scene changes (shot-changes). To analyze the performance of the system in this respect, we carry out a number of experiments, as explained below.

These experiments are performed with synthetic sequences used in [6, 7] slightly shifted between two subsequent fields to mimic (small) camera movements. A small movement is defined as a movement that shifts at a maximum of one quarter of the image per second. In the experiment, we use 60 images with a resolution of $320 \times 240$. The subsequent images are shifted by 3 pixels in vertical direction. The experimental results show that the threshold val-

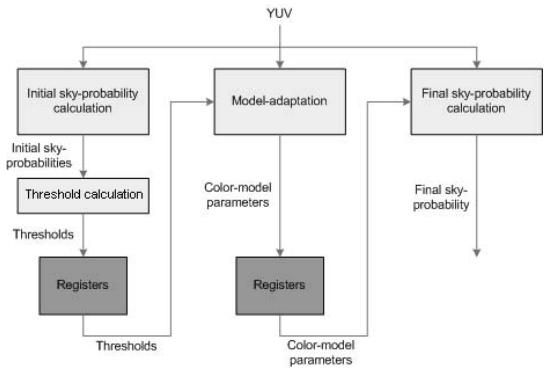


**Figure 2. Architectural overview**

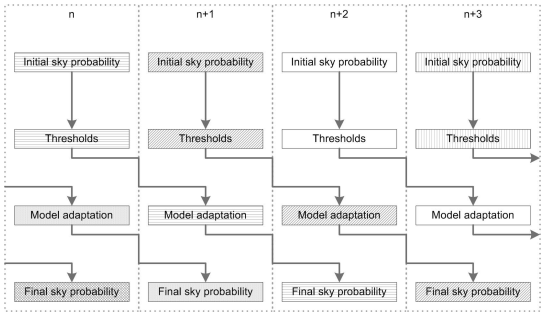


**Figure 3. Inter-field dependencies**

ues change very slowly (the thresholds only differ by 0.4% for small motions). Therefore, by using the previous field's thresholds, the color and vertical-position models changes slowly between the fields (by only 0.4% on all components). As a result, the difference between the original final sky probability and the sky probability based on delayed information is almost unnoticeable. A histogram of the difference in sky-probability values shows that more than half of the pixel values produce a difference of zero, and that 99.8% of all pixel values differs by only 2% at most. Furthermore, our experiment shows that the temporal incoherence of the resulting sky probability after scene changes (caused by using 2-field old information) are hardly noticable to human eyes, when the sequence is played in real-time.

The above considerations lead to an architecture as depicted in Fig. 2, where the large blocks on the top correspond to the three main stages of the algorithm, the threshold calculation is shown as a separate block, and the green blocks represent the registers that save the inter-field data, namely the threshold values and the model parameters. Using this architecture, the inter-field dependencies are illustrated in Fig. 3 (each chain is represented by a different color and fill). This figure shows that the final sky probability calculation for field *n+2* is based on the model parameters calculated for field *n+1*, and these parameters are based on the thresholds calculated in field *n* (the chain containing blocks filled with green horizontal lines).

In addition, we choose to do the initial sky probability calculation and the model-adaptation stage on a downscaled version of the original image. This expands the effective size of the texture analysis window, and reduces the amount of computations and required memory, as explained in 2.1.

### 3.2 Initial sky probability calculation

In the initial sky probability calculation stage, the architectural considerations are related to the texture probability calculation. The texture calculation block computes the Sum of Absolute Differences (SAD) and gradient of each pixel in the image, which are further appropriately scaled. The SAD is calculated over (vertically and horizontally) adjacent pixel pairs in a $5\times5$ window. The horizontal gradient is computed by taking the difference between the average values of the left and right halves of the analysis window. Similarly, the vertical gradient is computed by using the up and down halves.

The calculated gradient and SAD are then used as an index for a look-up table that represents the Gaussian weighting function used in [6, 7]. Implementing the additions and subtractions that are involved in this block (about 111 additions/subtractions) using a naïve approach, in which the operations are assigned dedicated adder blocks, utilizes more than 3,500 logic elements (35%) of the targeted FPGA.

We decided to optimize this by scheduling and resource binding, to reduce the large hardware utilization of this block. The scheduling and resource binding is performed according to the DeMicheli method [2]. The scheduling is done under resource constraints to find area/latency Pareto points, where the maximum latency is 26 clock cycles (because this is the number of cycles between two downscaled pixels in Standard Definition input video format assuming a system clock frequency of 100 MHz). Using manual scheduling, binding the resources and sharing the registers we obtained an optimization of the hardware costs for this block from more than 3,500 to only 1,056 logic elements.

The scheduling is performed as follows. We define two different building blocks: an adder block and a block that calculates the absolute difference (ABS-block). First, we determine the minimum number of blocks that calculate the absolute differences for the texture calculation. Second, we set the minimum number of adders to keep the maximum latency within its constraints. We determine that the minimum number of these block pairs is 4, and the minimum register utilization is 12 (using the minimum number of blocks and register sharing). The exact scheduling of the blocks using the ASAP algorithm described in [2], is

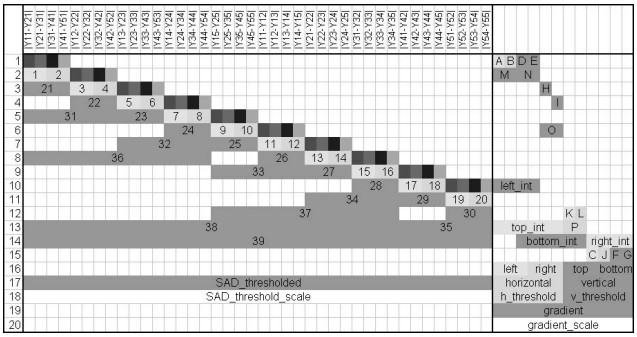


**Figure 4. Initial texture analysis scheduling**

**Table 1. Fig. 4 acronyms meaning**

| acronym | meaning | acronym | meaning |
|---|---|---|---|
| A | $Y_{11} + Y_{12}$ | N | D + E |
| B | $Y_{21} + Y_{22}$ | O | H + I |
| C | $Y_{31} + Y_{32}$ | P | K + L |
| D | $Y_{41} + Y_{42}$ | left_int | M + N |
| E | $Y_{51} + Y_{52}$ | right_int | O + P |
| F | $Y_{13} + Y_{23}$ | top_int | M + O |
| G | $Y_{43} + Y_{53}$ | bottom_int | N + P |
| H | $Y_{14} + Y_{15}$ | left | left_int + C |
| I | $Y_{24} + Y_{25}$ | right | right_int + J |
| J | $Y_{34} + Y_{35}$ | top | top_int + F |
| K | $Y_{44} + Y_{45}$ | bottom | bottom_int + G |
| L | $Y_{54} + Y_{55}$ | horizontal | right - left |
| M | A + B | vertical | bottom - top |

depicted in Fig. 4. The colors represent the resource binding (each color represents a different building block). The ABS-blocks are represented by the colors red, purple, blue and orange, the adders are represented by yellow, tan, green and turquoise, and the different time slots are represented in the vertical direction.

On the left side of Fig. 4, the index for the SAD look-up table is calculated. The absolute differences between the adjacent pixels are computed at the top of each column using the ABS-blocks. These are summed up using the adders numbered from 1 to 39, the results of the blocks on the upper side of the adders are used as the two terms in the summation. The resulting sum (of adder 39) is thresholded to form the $SAD\_thresholded$. The thresholding is needed to suppress the noise as explained in [6, 7].The $SAD\_thresholded$ is scaled for the correct index for the look-up table.

On the right side of this diagram, the index for the gradient look-up table is calculated. Tab. 1 shows the meaning of the scheduled additions that eventually lead to the two gradients, denoted as $horizontal$ and $vertical$. These two results are then thresholded and summed for the total $gradient$. Subsequently, the $gradient$ is scaled to obtain the correct LUT index.

### 3.3 Model adaptation

The model adaptation stage includes 1) the computation of the two thresholds for calculating the color model and vertical-position model, 2) computing a sky confidence metric, and 3) the actual calculation of the mentioned models. First, we consider the calculation of the two thresholds, and the sky confidence metric. This contains the calculation of the histogram and the cumulative density function (CDF) of the initial sky probabilities (see [6, 7] for details). The histogram of the initial sky probability is calculated, using a memory array, by incrementing the memory location corresponding to the value of the sky probability of each pixel, as soon as the sky probability is computed. During the blanking period of the video signal, the histogram values are integrated and weighted to compute the weighted CDF. The blanking period of the video signal is the period of time between the last pixel a field and the first pixel of the following field. The two thresholds and the sky-confidence factor are extracted from the weighted CDF. The creation of the CDF and the extraction of the mentioned parameters are performed during the blanking to accomodate a more relaxed computation requirement.

### 3.4 Final sky probability calculation

In the final sky probability calculation stage, the design considerations are related to the timing constraints. The pixels need to be processed within 8 system clock cycles (for an assumed system clock frequency of 100 MHz), because this is the amount of time between horizontal adjacent pixels in Standard Definition format video. The color probability in this final stage is computed by a subtraction (pixel color – mean color prescribed by the color model), a division (by the variance of each color component) and an access to a look-up table. As the variance is not fixed, an implementation pointed out that the division cannot be performed within 8 clock cycles. This timing problem is solved by using multiplication and bit-shifting instead of a division. We shift by a fixed number of bits and the multiplication factor is determined by examining the variance (prescribed by the color model). As a result, the color probability is computed by a subtraction, a multiplication, a bit-shift and an access to a look-up table, which can be performed within 8 clock cycles.

**Table 2. HW resource utilization**

| Resource utilization | Logic Elements LUT | Logic Elements Registers | Memory bits | DSP blocks |
|---|---|---|---|---|
| Total resources | 10,570 | | 920,448 | 48 |
| Algorithm | 2,710 | 3,046 | 23,687 | 13 |
| (relative) | 54.4% | | 2.6% | 27% |
| Complete system | 3,236 | 4,182 | 33,927 | 13 |
| (relative) | 70.2% | | 3.7% | 27% |

### 3.5 Implementation results

The proposed architecture was implemented on an Altera EP1S10 FPGA and was found to achieve a maximum system clock frequency of 104 MHz. For the sake of ease of calculations and to have some safety margins for the PLL, we set the system clock frequency at 100 MHz. The device contains DSP blocks that are 9-bit arithmetic units and can perform multiplication, additions, subtractions and/or accumulations. Table 2 depicts the hardware resource utilization of our implementation. In total, 54.5% of the available logic elements, 2.6% of the available memory bits and 27% of the available DSP blocks are used for the algorithm only (excluding the I/O infrastructure for camera and output (VGA) handling), and 70.2%, 3.7% and 27%, respectively, are utilized for the complete system (including the I/O infrastructure).

## 4 Experimental results

We carried out experiments for functional verification and performance evaluation of the proposed sky-detection algorithm implementation. The hardware framework for these experiments is as follows. The hardware implementation platform incorporates an Altera Stratix EP1S10 FPGA that runs at 100MHz, the Omnivision OV7620 camera that provides the input video image, and a VGA display for the final sky probability output. The camera can deliver video at a resolution of 640×480, with fixed frame rates between 0 – 30 fps. In the video-input link, the FPGA is the slave and the camera is the master. The FPGA receives the image in a pixel by pixel fashion and therefore the computations are also done pixel by pixel. Using this framework and a video input with a resolution of 640×480, the correct working of the implementation for video streams of up to 30 fps is verified.

The mode of the system operation can be split into two parts: operation during the active video and operation during the blanking of the video. Because of the pixel-synchronous nature of the system, the pixel frequency defines the maximum available time in the critical path. The critical path in our system is the final sky probability calculation block, as this stage runs at the original pixel frequency (the rest of the blocks use a scaled version of the input image). The pixel processing time is computed to be 8 clock cycles of 100 MHz, which is equal to 80 ns. During the blanking of the video, several blocks in the model-adaptation stage are active, this contains the *calculate adaptive threshold and sky confidence* block, the *adapt color model* block, and the *adapt vertical-position model* block. These blocks perform their computations during the blanking, because they need to wait until all pixels are processed and the results need to be ready before the first pixel of the following field is received. The *adapt color model* block and the *adapt vertical-position model* block only perform the normalization during the blanking, whereas the *calculate adaptive threshold and sky confidence* block perform the integration and multiplication of all sky probabilities during this period of time. Therefore, we only need the time of the *calculate adaptive threshold and sky confidence* block to compute the minimum blanking time. This takes 2304 clock cycles. The theoretic number of cycles needed for processing one frame $C'_{frame,hw}$ can be calculated as:

$$C'_{frame,hw} = (dim_x \times dim_y \times C_{pixel}) + (2 \times C_{blanking}),$$

where $dim_x \times dim_y$ is the image resolution, $C_{pixel}$ is the number of cycles needed for each pixel and $C_{blanking}$ is the number of cycles needed during the blanking period (in our case it is for the adaptive threshold and sky confidence calculation). When using $C_{pixel}$ = 8, $C_{blanking}$ = 2304 and a resolution of 640×480, this results in a $C'_{frame,hw}$ of 2,462,208 clock cycles, which is equal to 24.6 ms. Thus, a maximum frame rate of up to 40 fps can be processed by our implemented hardware system. The theoretical number of cycles for one frame $C'_{frame,hw}$ can be used to estimate the active duty cycle of our implementation (the percentage of time that the hardware is actually processing) for the validated frame rates, as this is the number of cycles that the system is actually operational. The duty cycle $\tau$ can be calculated by

$$\tau = \frac{C'_{frame,hw}}{f_{clk}} N_{frames},$$

where $f_{clk}$ is the system clock frequency (100 MHz) and the $N_{frames}$ is the frame rate. The duty cycles resulting from our experiment are listed in Tab. 3.

As stated before, we verified the correct working for frame rates of 30 fps. At this frame rate, Tab. 3 shows that the hardware implementation is only active for 74%. Furthermore, this means a pixel and frame accurate probability-map for the sky area can be created in real-time for Standard Definition video images.

The most critical parts of our hardware implementation are the scaling block (this block downscales the image for

**Table 3. Duty cycles for images of 640×480**

| frame rate | active duty cycle $\tau$ |
|---|---|
| 10 fps | 0.25 |
| 15 fps | 0.37 |
| 20 fps | 0.49 |
| 25 fps | 0.62 |
| 30 fps | 0.74 |

the initial sky probability calculation and model-adaptation stage) and the final sky probability calculation block, because these blocks operate on the full image resolution. All other blocks operate on the downscaled version of the image for the creation of the adaptive model (QCIF resolution), so these blocks can process images of various resolutions with a constant speed. For higher resolution input images, the downscale factor is set higher to retain a constant analysis resolution. The scaling block is not computional intensive compared to the final sky probability calculation block. In the final sky probability calculation, the most critical parts are the $P_{color}$ and $P_{sky}$ calculations. The $P_{position}$ is only calculated once per line and this calculation can be done in advance (before each line starts), and therefore, this block is less critical. The $P_{color}$ computations consists of subtractions, multiplications, bit-shifts and look-up table accesses, as stated in Section 3.4, and the $P_{sky}$ calculations consists of multiplications only. These computations are based on 8-bit integers.

When the final sky-detection stage is further pipelined, one single 8-bit multiplier determines the speed of the system, as the subtractions, bit-shifts and look-up table accesses are performed faster. The pipelining of this stage can be performed by adding registers to remember all the intermediate results in the $P_{color}$ and the $P_{sky}$ blocks. In such pipelined implementation, the time needed for the multiplication is at most 7.025 ns (retrieved from Quartus II), which means that the rate at which new pixels can be processed is 142 MHz. This theoretical rate exceeds the pixel rate of the highest HDTV quality images, which is 74.25 MHz for a resolution of 1920×1080, significantly, indicating that our design can be used for HDTV format video material with minor modifications.

## 5 Conclusions and future work

In this paper, we have presented a hardware architecture for a real-time implementation of a sky-detection algorithm. The algorithm includes a 2-d texture and gradient analysis, an adaptive thresholding method, and an adaptive color model creation. This implementation uses only 54.4% of the Altera EP1S10 FPGA and can operate at clock frequencies up to 104 MHz.

The proposed architecture employs pipelining and scheduling and resource binding to enable an efficient real-time implementation of the algorithm. Furthermore, parallel execution of different computational blocks were made possible due to removing calculation dependencies, by reusing some information calculated in previous fields.

Our experiments have shown that the system can process Standard Definition format video material in real-time. As indicated by our calculations, after some minor modifications, video streams with a maximum pixel rate of 142 MHz should be processable. This means that it should be possible to handle HDTV video material with a further optimized system.

Due to the limited duration of the project, we aimed for a simplified proof of concept using a simplified algorithm to provide insight in the feasibility of such algorithm for real-time operation on TV signals. The implementation and analysis of the original algorithm is left for future work.

## Acknowledgments

The authors would like to thank Eric van der Laan, Marco Pas and Eddy de Ridder of LogicaCMG's Working Tomorrow program for creating the graduation project. In addition, we would like to thank Eric van der Vliet and Peter de With of LogicaCMG for the project definition and for guidance during the project. We would also like to thank Dirk Piepers and Erwin Dewitte of Philips Consumer Electronics for their cooperation for setting up the project.

## References

[1] Altera Corporation, 101 Innovation Drive, San Jose, CA 95134. *Nios development board, reference manual, stratix edition*, 1.2 edition, December 2003.

[2] G. DeMicheli. *Synthesis and Optimization of Digital Circuits*, chapter 4-6, pages 146–245. McGraw-Hill, 1 edition, January 1994.

[3] A. C. Gallagher, J. Luo, and W. Hao. Improved blue sky detection using polynomial model fit. In *ICIP*, pages 2367–2370, 2004.

[4] S. Herman. Real-time segmentation of video image sequences—part 2: Sky detection, March 2002.

[5] S. Herman and E. Bellers. Locally-adaptive processing of television images based on real-time image segmentation. In *International Conference on Consumer Electronics*, pages 66–67, 2002.

[6] B. Zafarifar and P. H. N. de With. Adaptive modeling of sky for video processing and coding applications. In *27th Symposium on Information Theory in the Benelux*, 2006.

[7] B. Zafarifar and P. H. N. de With. Blue sky detection for picture quality enhancement. In *Advanced Concepts for Intelligent Vision Systems (Acivs 2006)*, 2006.