An Efficient Architecture Solution for Low-Power Real-Time Background Subtraction

Hamed Tabkhi
Department of Electrical and Computer Engineering
Northeastern University
Boston (MA), USA
Email: tabkhi@ece.neu.edu

Majid Sabbagh
Department of Electrical and Computer Engineering
Northeastern University
Boston (MA), USA
Email: msabbagh@ece.neu.edu

Gunar Schirner
Department of Electrical and Computer Engineering
Northeastern University
Boston (MA), USA
Email: schirner@ece.neu.edu

***Abstract*—Embedded vision is a rapidly growing market with a host of challenging algorithms. Among vision algorithms, Mixture of Gaussian (MoG) background subtraction is a frequently used kernel involving massive computation and communication. Tremendous challenges need to be reolved to provide MoG's high computation and communication demands with minimal power consumption allowing its embedded deployment.**

This paper proposes a customized architecture for power-efficient realization of MoG background subtraction operating at Full-HD resolution. Our design process benefits from system-level design principles. An SLDL-captured specification (result of high-level explorations) serves as a specification for architecture realization and hand-crafted RTL design. To optimize the architecture, this paper employs a set of optimization techniques including parallelism extraction, algorithm tuning, operation width sizing and deep pipelining. The final MoG implementation consists of 77 pipeline stages operating at 148.5 MHz implemented on a Zynq-7000 SoC. Furthermore, our background subtraction solution is flexible allowing end users to adjust algorithm parameters according to scene complexity. Our results demonstrate a very high efficiency for both indoor and outdoor scenes with 145 mW on-chip power consumption and more than 600x speedup over software execution on ARM Cortex A9 core.

I. Introduction

Among high-performance low-power applications, embedded vision is considered as one of the fastest growing areas in the incoming decades [1], [2]. While vision computing has already received much research effort, embedded deployment of vision algorithms is still in early stages. The interest in high-performance, low-power vision computing is high with exciting markets such as Advanced Driver Assistance System (ADAS), industrial vision, and video surveillance. Approaching demanding high-performance low-power solutions introduces tremendous challenges to embedded architects. For this, the general trend is toward heterogeneous solutions combining general-purpose processors with specialized accelerators; compute-intense vision kernels execute on accelerators, and remaining portions execute on processors.

Among vision kernels, background subtraction is considered as a primary vision kernel for separating foreground pixels (i.e., moving objects) from a static background scene [3]. Background subtraction algorithms range from history-based realizations to adaptive learning algorithms. Among them, Mixture of Gaussian (MoG) is an efficient and frequently used algorithm for scenes with static camera position [4]. MoG is an adaptive algorithm where multiple Gaussian components track the background for each pixel. Realization of MoG at Full-HD (1920*1080 resolution) involves massive computation and communication, which result in very high power consumption. Our initial profiling hints that 610 ARM Cortex-A9 cores would be required for real-time MoG in Full-HD with 60 frames per second, rendering a SW approach infeasible. Hence, customized architecture solutions for managing the immense computation and communication are needed to provide the necessary compute power.

Few approaches have been already proposed specialized solutions for embedded realization of MoG background subtraction [5], [6]; mainly targeted for FPGAs. On the flip side, the existing solutions operate on a very limited resolution (300*200); do not efficiently tackle the high computation and communication demands of MoG. Furthermore, the existing solutions offer almost no flexibility to support variety of scenes with varying complexities. Simultaneously, the current market demands much higher resolutions (e.g. Full-HD 1920x1080, or SXGA- 1280x960) with high frame rates (e.g. 60Hz), as well as flexible solutions that are deployable for a wide range of scenes (from simple indoor to complex multi-motion outdoor scenes). Novel specialized architectures are needed that satisfy the market demands: a low power implementation of MoG operating in Full-HD with adjustability for scene complexity.

In this paper, we propose a specialized architecture solution to realize Full-HD MoG-based background subtraction for embedded deployment. At the same time, we highlight how ESL design principles can help to manage design complexity and reduce the design and validation time. We start with an executable specification model captured in a System-Level Design Language (SLDL) to derive our architecture and finally the handcrafted RTL design. We have prototyped our proposed MoG solution on Zynq-7000 SoC. The solution consists of 77 pipeline stages operating at up to 148.5 MHz processing Full-HD 60Hz inputs with only 145 mW power consumption. In addition, our solution offers flexibility to the end users for adjusting the algorithm parameters with respect to the scene complexity. Our experimental results demonstrate the benefits of our solution for both indoor and outdoor scenes.

This paper is organized as following: Section II overviews relevant prior work. Section III briefly provides background and additional motivation. Following that, Section IV describes our proposed solution for MoG background subtraction. Section V presents the results of our MoG implementation. Finally, Section VI concludes the paper and touches on future work.

II. Related Work

Only few embedded solutions for MoG background subtraction have been proposed [5], [6], [7], [8]. [6] designs a single stage MoG implementation on Xilinx Virtex-II Pro V2P30 FPGA. The architecture in [6] avoids using the costly square root while accepting a significant loss in quality. Overall, by targeting frame size of 320*240 at 30 frames per second, [6] operates at 27.65MHz with 265mW power consumption. Very similar, the approach in [7] only performs on a very limited resolution with 120*120.

[5] deploys MoG onto a Xilinx Virtex-II XC2v6000. The proposed architecture includes six pipeline stages and supports 640*480 resolution. [5] starts from MoG with single Gaussian and then extends to two Gaussian components. [5] also enhances MoG algorithm to improve the quality of scenes with light variations. In a different approach, [8] aims for full-HD resolution. However, [8] remains at simulation-level without an actual execution in real-time embedded deployment. The reported results are estimations from the synthesis tool. With focusing on simulation, [8] does not validate the applicability of the approach in a real-time environment.

In comparison to the earlier approaches, we present in this paper a solution that targets a much higher resolution (Full-HD at 60Hz). Our FPGA realization keeps the execution semantic of the original MoG algorithm to minimize the quality loss. Our FPGA realization only adjusts operation width sizes to reduce computation complexity and performs the Gaussian parameter compression to reduce memory traffic. We also utilize quality assessment tools to quantify our design correctness compared to the original algorithm.

III. Background and Motivation

In order to place our proposed solution into perspective, this section first briefly introduces the algorithm for MoG background subtraction. Following that, it motivates the need for a customized solution by investigating on MoG execution on embedded processors.

A. Mixture of Gaussian algorithm

Among the background subtraction algorithms, Mixture of Gaussian (MoG) is widely used for deployments with fixed camera position. MoG offers a very good quality and efficiency in capturing multi-modal background scenes [9], [10]. Algorithm 1 outlines the reference MoG algorithm (see further detail in [11]). The algorithm loops through all pixels in the frame (lines 2).

For each pixel, the algorithm first classifies the pixel's Gaussian components into *match* or *non-match* components (line 6). A component matches, if the component's *mean* is within a match threshold Γ_{FG} of the current pixel value. Gaussian parameters are updated based on *match* classification. In case that no match exists (line 16), the algorithm creates a new Gaussian component, called virtual component replacing

Algorithm 1 MoG Algorithm
```
1: function MoG (in Frame, inout Gaussian, out Foreground)
2:     for $i = 0$ to $numPixel$ do
3:         $match = 0$
4:         for all $gaussian$ do
5:             $diff[k] = abs(m_k - pixel)$
6:             if $diff[k] < \Gamma_{match}$ then                ▷ Match
7:                 update $\{w_k, m_k, sd_k\}$ based on $\alpha$
8:                 $match = 1$
9:             else                                              ▷ non-Match
10:                update $\{w_k\}$
11:            end if
12:        end for
13:        for all $gaussian$ do
14:            normalization $w_k$
15:        end for
16:        if $!match$ then
17:            Create virtual component for the smallest $w_k$
18:        end if
19:        for all $gaussian$ do
20:            Rank and Sort $gaussian$ based on $w[k]/sd_k$
21:        end for
22:        $Foreground = 1$
23:        for $k = HighestRank$ to 0 do
24:            if $\omega_k \geq \Gamma_{FG}$ && $diff[k]/sd_k < \Gamma_{match}$ then
25:                $Foreground = 0$
26:                break
27:            end if
28:        end for
29:    end for
30: end function
```

the Gaussian component with smallest *weight* value (line 17). Then, the components are ranked and sorted based on their *weight* over *standard deviation* ratio (line 19). Starting from the highest rank component, the algorithm declares a pixel to be background if its weight is less than the FG threshold Γ_{FG} and the updated *mean* over updated *sd* is less than match threshold Γ_{match} (line 23). When finding the first match, the comparison stops, and the algorithm continues with the next pixel. MoG demands significant computation with many conditional statements (if-then-else) for calculating the foreground state of an individual pixel.

B. MoG Performance analysis

To assess the feasibility of a MoG Software (SW) solution, we have implemented the MoG algorithm (Algorithm 1) in C and compiled it for ARM Cortex A9 using gcc with $O3$ optimizations and enabled NEON optimizations. Running the code on one ARM core of Zynq platform, requires 610s (360s) for 60 frames in full-HD (half-HD). In other words, this execution is 610x and 360x slower than real-time. Even if real-time execution could be achieved by parallel execution on 610 (360) A9 cores, power consumption would be unacceptably high. Assuming each core (with cache) consuming 0.6 W at 666 MHz [12], 360 W (216 W) are required for full-HD and half-HD respectively. Hand-crafted optimizations with NEON SIMD also do not put this approach into the solution envelope. Even with an unrealistic 4x SIMD speedup, still 152 (90) A9 cores are needed, which exceeds power constraints. Another possible approach is MoG implementation on CGRAs-like structure, such as the one proposed by [13]. CGRAs can achieve better performance / power efficiency compared to SW-only execution, however their efficiency is still lower then a customized solution.

MoG also imposes a high computation demand to system for fetching and writing back the Gaussian parameters (*mean*, *weight* and *standard deviation*). Assuming 32bit parameter length per Gaussian parameters and 3-5 Gaussian components per pixels, 72 to 120 Bytes of memory accesses per pixel are required for updating Gaussian parameters. All Gaussian parameters together require 74 MB of memory for Full HD (1920 * 1080) resolution. This by far exceeds the size of on-chip memory in embedded platforms. With updating on every pixel, MoG creates a memory bandwidth of 8 GB/Sec. and 4 GB/Sec. for processing Full-HD frames at 60 Hz and 30 Hz respectively.

Overall, looking at the both computation and communication demands of MoG, a customized architecture solution is needed to power efficiently realize Full-HD MoG in real-time. Many additional challenges remain such as maintaining flexibility to support market diversity.

IV. MoG Specialized Architecture

This section introduces our proposed architecture solution for efficient realization of MoG background subtraction for real-time vision processing.

A. MoG Design flow

In order to realize our proposed hand-crafted MoG solution, we follow a system-level design flow throughout the entire design process. The ESL flow allows us to tackle the design complexity, dividing it into manageable steps. Fig. 1 captures the essence of our design flow starting from MoG specification down to actual execution. Individual MoG models (green background color) include: *Specification*, *Architecture*, *Implementation* and *Execution* models. In between, the refinement/synthesis tools (or manual design) realize MoG in the next lower abstraction level. For each level, validation tools (yellow background color) validate design properties including functionality, quality, and timing.

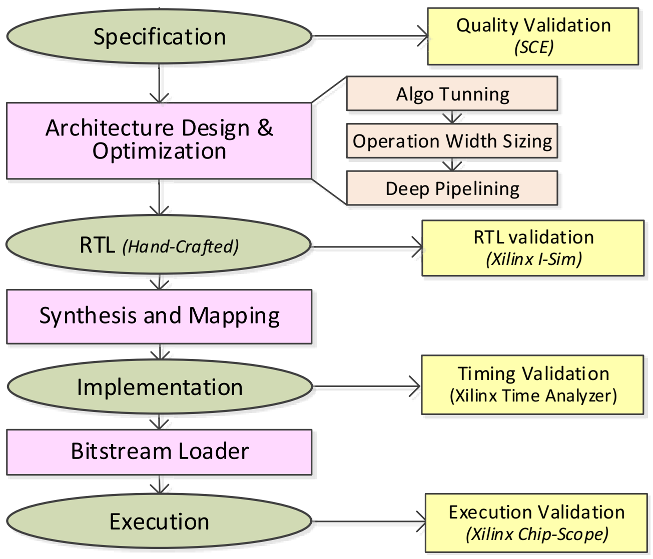


Figure 1: FPGA-based MoG Design Flow.

The specification model is our golden model that reflects all system-level design decisions. Any additional decisions triggered by lower levels (e.g., RTL level) are first validated at specification level. To construct our specification model, we have captured MoG algorithm (Algorithm 1) in the SpecC System-Level Design language (SLDL) [14].

Fig. 2 illustrates our initial MoG specification captured in SpecC SLDL including coarse-grained parallelism, pipeline stages and communication challenges. The pipeline stages are: *Gaussian Update*, *Weight Normalization* and *FG Detection* operating on streaming pixels. In *Gaussian Update,* (line 4 to line 12 of Algorithm 1), Gaussian components execute in parallel independently from each other. *Weight Normalization* is a synchronization point for normalizing the updated weight parameters (line 13 to line 15 of Algorithm 1). Finally, *FG Detection* determines the FG/BG status of the pixel (line 16 to line 28 of Algorithm 1). The input channels are *Gaussian parameters* (read from memory) and *Gray pixel* streamed from camera. The output channels are *FG mask* streamed out to output and *Gaussian parameters* for writing back the updated Gaussian values.

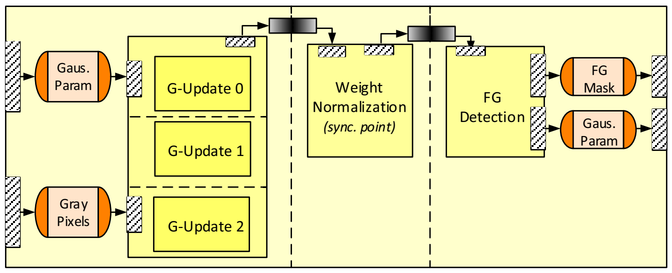


Figure 2: MoG specification model including coarse-grained parallelism.

B. MoG Architecture

As the architecture exploration tightly depends on the timing analysis of the design, we combine the architecture exploration with the process of RTL design, synthesis, and implementation. Following the design flow in Fig. 1, we iteratively refine the MoG architecture with respect to MoG specification model. Our proposed MoG architecture targets the Zynq-7000 XC7Z020-CLG484-1 platform which combines a dual-core Cortex A9 as a Processor System (PS) and Programmable Logic (PL) in a single chip [15]. However, the principles and solutions are similarly realizable on other FPGA devices as well as ASIC implementation.

Fig. 3 outlines the primary elements of our proposed architecture template. It consists of two clock domains: the pixel clock domain is driven by HDMI clock frequency, and Gaussian parameter clock domain driven by the AXI stream clock. The MoG algorithm kernel executes efficiently over the pixel stream. At the same time, Gaussian parameters are accessed through the AXI stream connected to the memory controller. We propose dedicated ports for accessing pixel stream and Gaussian parameters. The ports for accessing the pixel stream is directly connected to the system interface (e.g., camera or monitor). Gaussian parameters are accessed by dedicated DMA channels for burst Read/Write access independently from the

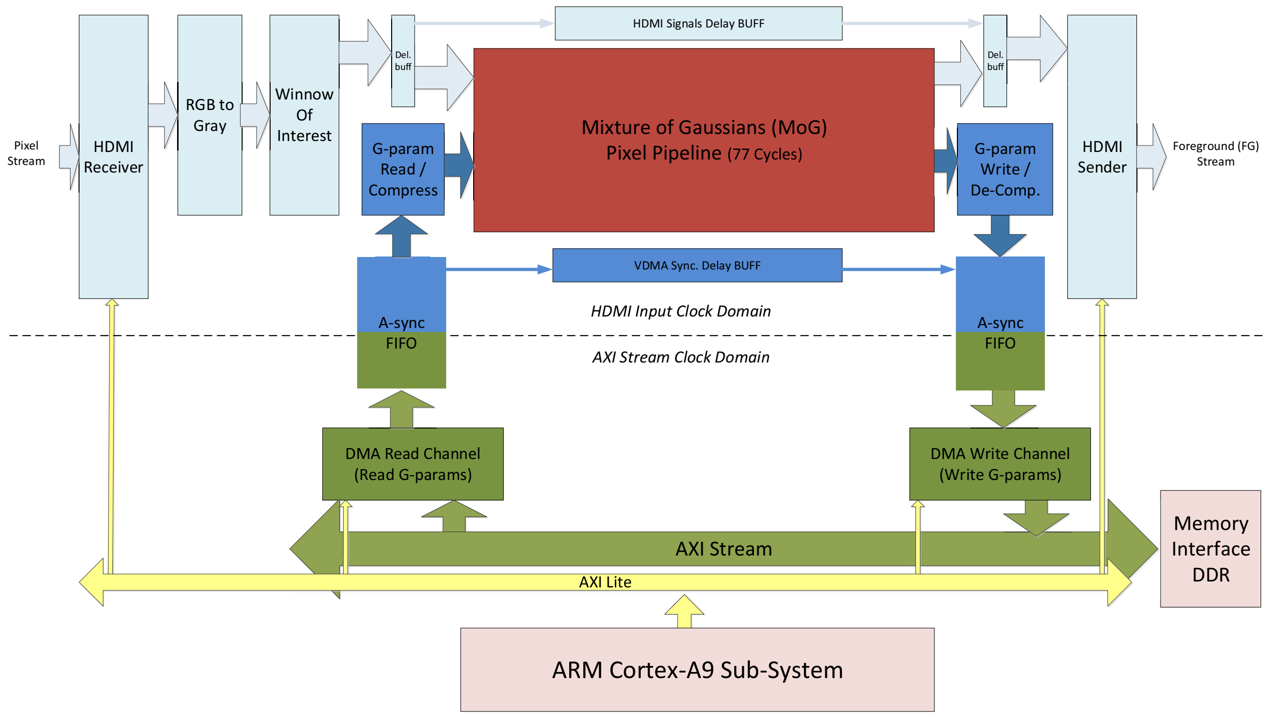


Figure 3: MoG architecture overview.

streaming pixels. DMA channels are connected to the system memory controller through the streaming buses. The DMA channels operate in circular modes; they continuously read/write back Gaussian parameters. The DMA channels are working in parallel; however, they are synchronized to each other to preserve correct read after update sequence.

The key system challenge is the synchronization between streaming pixels and their associated Gaussian parameters. The DMAs must guarantee that when new pixel is available, the corresponding Gaussian parameters will be available at the same time. Our architecture solution achieves this by proposing asynchronous FIFOs for bridge the two clock domains and compensating for the bandwidth mismatch in supplying operational data.

We have realized the proposed architecture (highlighted in Fig. 3) through manual RTL implementation guided by the system-level specification. Alternatively, High-Level Synthesis (HLS) tools, e.g., Xilinx Vivado, could be employed. They are promising especially when a system specification model is available. However, compared to hand-crafted design, HLS are typically less efficient. Furthermore, for the case of MoG targeted at FULL-HD resolution, HLS tools are unable to meet the timing requirements for 148.5 MHz clock frequency. In result, we chose the hand-crafted approach using Verilog HDL to capture our RTL model. In the following, this article explains the different techniques employed to make possible the realization of our architecture solution operating at 148.5 MHz clock frequency.

C. *MoG Algorithm Tuning*

One important optimization step is MoG algorithm tuning to match it better to hardware executions. As an example, ranking and sorting in the reference MoG algorithm (line 19 of Algorithm 1) helps in software execution to eliminate some computation at foreground detection. A Gaussian component with a higher rank is more likely to match the current pixel, in result the loop can be terminated early. Conversely, due to the serial execution behavior of ranking and sorting they significantly increase the pipeline critical paths in HW.

To optimize for HW execution, we replace ranking and sorting with an unconditional checking of all Gaussian components. Algorithm 2 highlights our proposed algorithm tuning only for the foreground detection stage. Compared to line 19 of Algorithm 1, Algorithm 2 completely removes the ranking and sorting part and compares Gaussian components unconditionally in parallel (note the order of finding a match does not matter).

Algorithm 2 MoG excerpt (no sort)

```
1: for $k = 0$ to $numGau$ do
2:     if $\omega_k \geq \Gamma_{FG}$ && $diff[k] < sd_k * \Gamma_{match}$ then
3:         $Foreground = 0$
4:         break
5:     end if
6: end for
```

In a different optimization, we also reduce the number of divide operations and replace them with multiply operations through algebraic manipulation. In comparison to divide, multiply operations have lower latency which can reduce the pipeline critical path. For example, we replace $diff[k]/sd_k < \Gamma_{match}$ conditional statement in reference

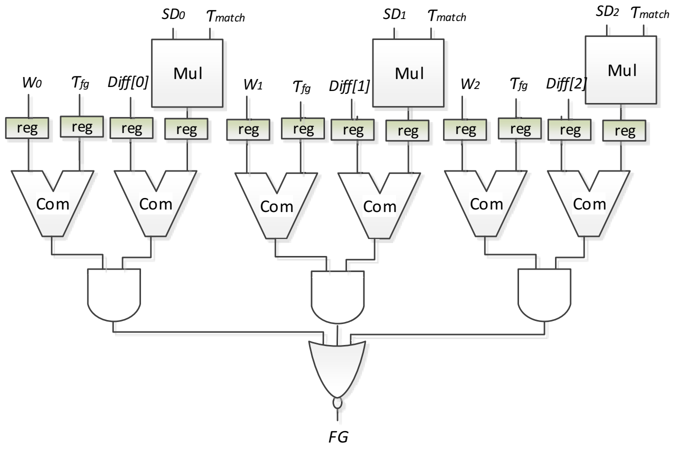


Figure 4: RTL realization of Match Detection.

algorithm with $diff[k] < sd_k * \Gamma_{match}$ (highlighted in Algorithm 2). By replacing divide operation with multiply in a conditional statement a considerable reduction in the critical-path is achieved. This also preserves the correct functionality of the algorithm. Fig. 4 demonstrates the hardware realization of proposed algorithm tuning. All Gaussian parameters are compared in parallel. Then, final FG/BG (1/0) result is a NOR operation within all Gaussian components.

D. Operation width sizing

One effective way to reduce the computation complexity and its associated delay is to identify the best width for arithmetic operations, as well as the optimum range of Gaussian parameters stored in off-chip memory. For the computation aspects, this poses a trade-off between computation delay and quality; higher operation width leads to higher quality in the cost of higher latency. For communication aspects, this poses a trade-off between the bandwidth and quality; higher Gaussian width leads to higher quality but also demands higher bandwidth. During the exploration, we assess the quality against the ground truth (Algorithm 1). We use Structural SIMilarity (MS-SSIM) metrics. MS-SSIM focuses on the structural similarity between two frames which is more similar to human perception [16]. MS-SSIM quantifies the quality as a value between 0 to 1 where a higher value means closer similarity.

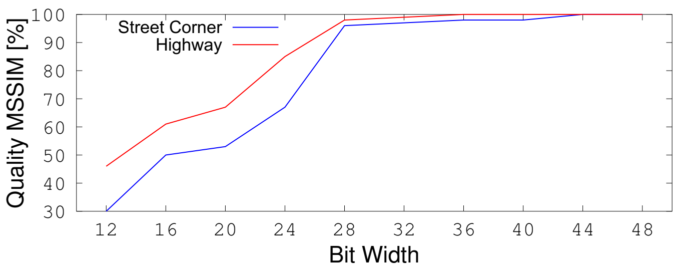
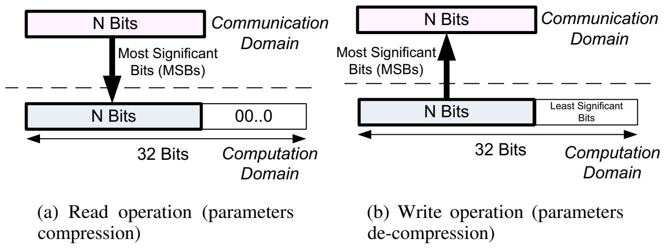


Figure 5: SQRT quality / operation width exploration.

In computation aspects, operation width sizing refers to identifying the optimal quality / width point which is highly desirable for high-latency arithmetic (SQRT, divide and multiply). With respect to our design flow, we first explore the quality / operation width trade-off at MoG specification level. Following that we updated the width changes in our RTL model. Fig. 5 exemplifies our exploration for SQRT operation in two different scenes: street corner and highway. The X-axis shows width of SQRT operation and the Y-axis the resulting quality (MS-SSIM) are compared to an ideal 48-bits SQRT. Overall, we observe an almost linear improvement in quality as SQRT width increases from 8-bit to 28-bit with. In contrast, very marginal improvement are achieved with wider than 28-bit operations. In result, we choose 28-bit as an optimal width for SQRT. Reducing to 28 bits has a minimal quality loss of 2% and 4% for highway and street corner respectively. Similar studies have been performed for divide and multiply operations resulting to 16-bit width multiply and a 24-bit width divide operations.

Two major units in our architecture solution are Gaussian parameters compression upon write and de-compression upon read accesses. We propose Gaussian compression / de-compression units in the AXI communication path between the asynchronous FIFOs and MoG kernel; illustrated in Fig. 3. Their computation overhead is fairly small; bounded to selecting Most Significant Bits (MSBs) of Gaussian parameters. Fig. 6 highlights the liner bit selection for Gaussian parameters. This article mainly focuses on the computation aspect of design. In [17], we have shown our system-level exploration of that aspect and have reduced the memory bandwidth requirements by 49% with a penalty of less than 1% in quality.

In a nutshell, 244 bits Gaussian parameters per pixel achieves 99% quality; note the full-length parameters require 480 bits Gaussian parameters per pixel. Applying the Gaussian compression / de-compression in SW implementation reduce execution time from 610s to 340s. This reveals that the ARM's memory interface is one of the major bottlenecks in SW execution. However, SW execution is still much slower than real-time requirements.



(a) Read operation (parameters compression)

(b) Write operation (parameters de-compression)

Figure 6: Gaussian parameters compression / de-compression.

E. Deep Pipelining

Deep pipelining is a common approach to increase operational frequency allowing architecture to achieve 148 MHz. To facilitate the pipelining process, we use a hierarchical pipelining with two levels of granularity: Macro pipelining and Micro pipelining. Macro pipelining realizes the coarse grain pipeline stages with independent functionality. Micro pipelining further breaks down individual Macro stages.

Macro pipelining identifies coarse-grained pipeline stages with fairly independent computations. At specification level, we have already identified three Macro pipeline stages. At

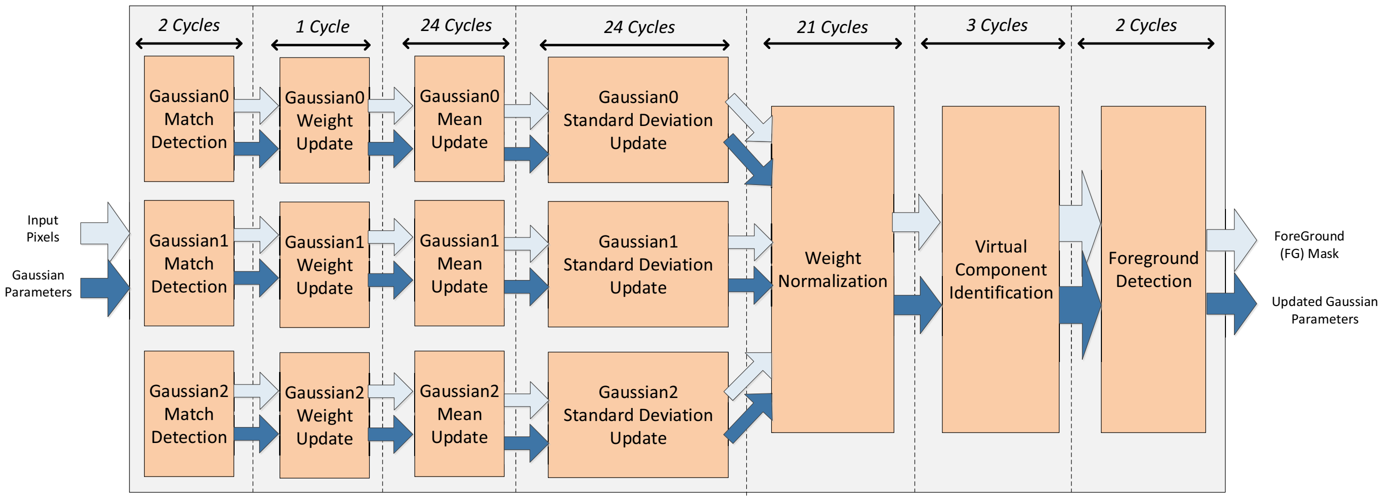


Figure 7: MoG pipeline hierarchy.

this level, we further increase the pipeline to 7 stages. Fig. 7 shows the extended Macro pipeline stages for MoG RTL model including: (1) *Gaussian Match Detection*; (2) *Weight Update*; (3) *Mean Update*; (4) *SD Update*; (5) *Weight Normalization*; (6) *Virtual Component*; (7) *Foreground Detection*. In the first four pipeline stages, all three Gaussian components execute in parallel.

In Micro pipelining, we break down the critical operations of individual Macro stages into smaller stages. As an example, Fig. 8 illustrates the micro-pipeline of *Mean Update* stage which is divided into 24 micro pipeline stages including divide (18 pipes), Multiply (4 pipes) and 2 more pipes for subtract and add operations. Intermediate pipeline registers are added between micro pipeline stage. For IP blocks with deep pipeline (divide and multiply) we add pipeline buffers to preserve current intermediate values. Similarly, FG detection stage (highlighted in Fig. 4) is divided into two micro pipelines by adding pipeline registers between multiply and threshold comparators.

The final architecture includes seven macro stages which are further divided into 77 micro stages (highlighted in Fig. 7). *Mean Update*, *Standard Deviation Update* and *Weight Normalization* are the stages with the highest computation loads internally slitted to 24, 24 and 21 pipes respectively.

F. MoG Software Control

Keeping flexibility is pivotal to support a wide set of markets and applications. To realize flexibility, we define a set of Memory Map Registers (MMRs) as part of the MoG solution. The MMRs are accessible by the ARM cores via the AXI-lite bus. Overall, we introduce 3 MMRs to adjust MoG algorithm parameters.

Fig. 9 highlights the MMRs including *Learning Factor* (α), *FG Threshold* (Γ_{FG}) and *Match threshold*. In a nutshell, *FG Threshold* determines the threshold for identifying a pixel as a foreground. *Match threshold* reflects the threshold for identifying a Gaussian component as a match component. *Learning Factor* represents the rate of updating Gaussian parameters. The registers enable the end users to adjust the

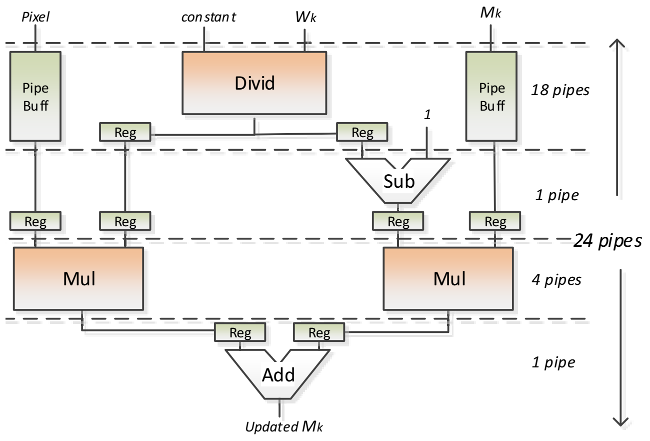


Figure 8: Micro pipelined Mean update.

MoG quality with respect to motion of foreground objects as well as scene complexity and conditions (e.g. indoor vs outdoor). For example, for a scene with faster-moving objects, higher learning factor required. Conversely, for slow moving objects, a lower learning factor is more demanded.

V. MoG Execution and Experimental Result

This section first introduces experimental setup. After that, it presents real-time execution and quality evaluations followed by reporting the FPGA resource usage and power consumption.

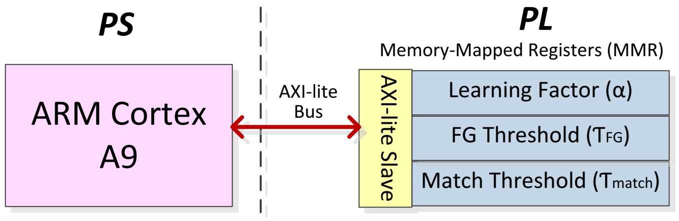


Figure 9: Flexibility support with MoG MMR registers

Table of Contents for this manuscript

A. Experimental Setup

Fig. 10 illustrates our experiment setup. We use the Avnet ZedBoard development kit which is based on the Zynq-7000 SoC XC7Z020-CLG484-1. We use two HDMI interfaces of a FMC-IMAGON extension card for video input and output. The MoG design operates on pixel stream received from the HDMI input interface. After calculating FG/BG status of individual pixels, the result (FG stream) is directed to the HDMI output for displaying it on an output monitor (highlighted in Fig. 10). When expanding the vision pipeline, the FG stream would remain in the Zynq platform for further processing either in FPGA or SW.

B. MoG real-time execution

To present execution results, a rich data-set representing real scenes at Full-HD resolution is required. For this purpose, we have recorded and collected our own input data-set including: indoor corridor, crowded highway and street corner. In a nutshell, indoor corridor represents a simple scene with a stable background with slow moving objects. Crowded highway represents an outdoor scene with very fast moving objects. Street corner shows a complex scene with multi-model changes in background including illumination and different FG moving objects with different speeds (pedestrians and cars).

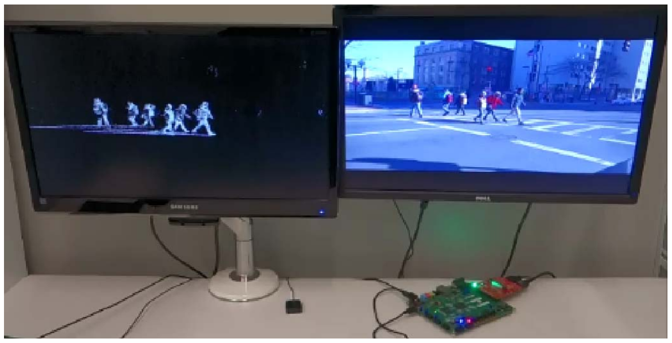
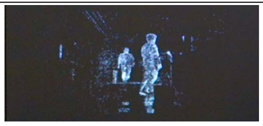
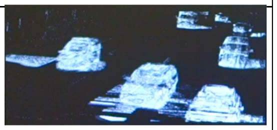
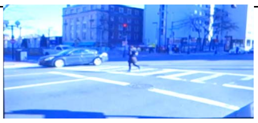
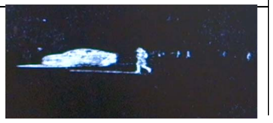


Figure 10: Experiment setup.

Table I shows one input frame per each scene as well as its corresponding FG mask output. For all three senses, we observe that moving foreground (FG) objects are correctly captured and exposed in FG mask output. Table I also highlights the algorithm knobs we have set for individual scenes. As an example, indoor corridor has the lower learning rate (α equal to 0.01) due to slow speed of moving objects. Conversely, α is much higher, 0.25 and 0.5 for the street corner and crowded highway respectively, as moving cars are much faster than humans. Other algorithm parameters (*FG Threshold* (Γ_{FG}) and *Match threshold*) are adjusted properly to deliver highest background subtraction quality.

Table I also highlights the algorithm knobs we have set for individual scenes. As our propose approach offers control to SW, end users can set algorithm knobs with respect to the scene complexity. As an example, indoor corridor has the lower alpha due to slow speed rate of moving objects which are pretty slow while alpha is much higher in crowded highway as moving cars are much faster than humans.

Table I: MoG execution on different scene complexity.

properties	input scene	FG mask
Indoor $\alpha : 0.01$ $\Gamma_{match} : 1.5$ $\Gamma_{FG} : 0.24$		
Highway $\alpha : 0.5$ $\Gamma_{match} : 2.5$ $\Gamma_{FG} : 0.24$		
Street $\alpha : 0.25$ $\Gamma_{match} : 1$ $\Gamma_{FG} : 0.18$		

C. Quality Evaluation

Evaluating the quality during FPGA execution is performed on an individual pixel level. We use chip-scope to record value traces of specific input pixels and their corresponding BG/FG status over consecutive executions of multiple frames. We developed a tool to compare the FG results gathered from chip-scope with MoG specification model. Due to limited RAM blocks in FPGA, only select pixels are recorded. Fig. 11 presents the input gray value variation of two sample pixels over 2000 frame execution. We considered one pixel as simple (blue line) as it has much lower variation over frames and one pixel as complex (red line) with many variations over frames.

Fig. 12 visualizes the comparison results for gray pixel variation (highlighted in Fig. 11). It compares FG decision gathered from chip scope with the golden model (SpecC simulation) for both simple and complex pixels over 2000 frames. Comparison results in Fig. 12 are based on the formula: $FG\_Mismatch = FG_{specification} - FG_{execution}$. Value 0 indicates that FG matches, while 1 / -1 (sharp edges in Fig. 12) indicate mismatches between HW execution and SpecC simulation. We observe less then 13 mismatches, resulting in 99.3% Mean Squared Error (MSE). For the complex pixel (Fig. 12b), we observe 32 mismatches (98.4% MSE). Further analysis has shown that the few mismatches are due to the different rounding of SQRT operation between SpecC simulation and RTL execution (even though using same bit width). Overall, we have compared the results of HW execution and SpecC simulation for more than 30 pixels over 2000 frames. On average, they show more than 99% similarity.

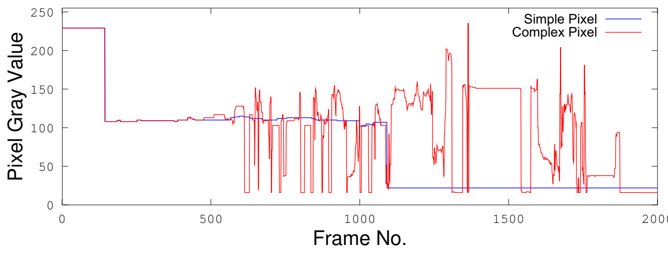
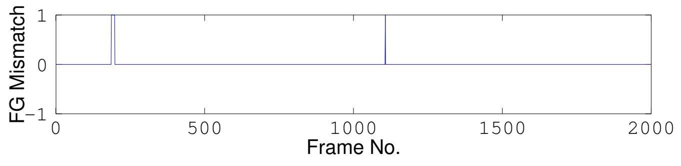
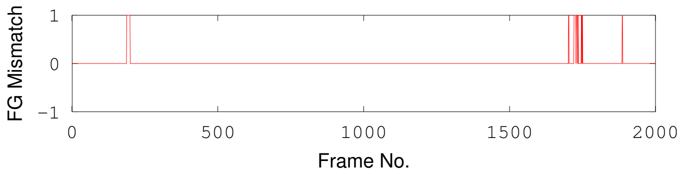


Figure 11: Pixel value distribution over 2000 frames.

Table of Contents for this manuscript



(a) SimplePixel



(b) Complex Pixel

Figure 12: Foreground Mask Value Comparison

D. Resource Utilization and Power Consumption

Table II lists the resource utilization of only the MoG design excluding system level components such as DMAs and HDMI interfaces. Our design occupies 38% of the slices, 8% of DSP blocks and 6% of 36-bit Block-RAMs of Programmable Logic.

Table II: System Resource Utilization

Resource	BUFG	DSP48E1	RAM Block	Slice	LUT	Register
Utilization[%]	9	8	26	38	20	12

Fig. 13 presents the power consumption of our MoG solution highlighting power contribution of individual Macro pipeline stages (as reported by the XPower Analyzer tool). Overall, the total on-chip system power consumption is 145mW with only 36mW is consumed by our MoG core. The remaining power (109mW) are consumed by the DMAs, AXI stream, and AXI lite buses as well as HDMI IN/OUT interfaces. Among the MoG macro stages, the Weight Normalization owns the majority of computation power budget (12mW) since it includes the heavy division operations.

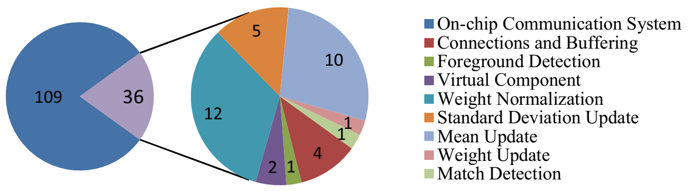


Figure 13: MoG power consumption

VI. Conclusions

In this paper, we introduced an architecture solution for MoG background subtraction at full-HD frame resolution. To aid in the design, we followed a system-level flow starting from MoG specification model down to MoG execution on FPGA. For realizing hand-crafted RTL design, we utilized a set of optimization techniques including algorithm tuning, operation width sizing and deep pipelining. The final MoG implementation consists of 77 pipeline stages operating at up to 148. MHz clock frequency. Furthermore, our architecture offers enough flexibility to end users for adjusting the algorithm knobs with respect to scene complexity. We implemented the proposed MoG architecture on Zynq-7000 SoC operating on Full-HD resolution HDMI input and producing HDMI output. The actual execution showed an almost flaw-less quality for both indoor and outdoor scenes (99% identical to simulation) with a power consumption of 145mW.

References

[1] J. Bier, “Implementing Vision Capabilities in Embedded Systems,” *Berkeley Design Technology Inc.*, Nov. 2012.

[2] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, “State-of-the-art in heterogeneous computing,” *Sci. Program.*, vol. 18, no. 1, Jan. 2010.

[3] S. ching S. Cheung and et al., “Robust techniques for background subtraction in urban traffic video,” *SPIE Electronic Imaging*, 2007.

[4] C. Stauffer and W. E. L. Grimson, “Adaptive background mixture models for real-time tracking,” in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2, 1999, pp. –252 Vol. 2.

[5] K. Appiah and A. Hunter, “A single-chip fpga implementation of real-time adaptive background model,” in *IEEE International Conference on Field-Programmable Technology*, 2005, pp. 95–102.

[6] J. Schlessman, M. Lodato, B. Ozer, and W. Wolf, “Heterogeneous mpsoc architectures for embedded computer vision,” in *IEEE International Conference on Multimedia*, 2007, pp. 1870–1873.

[7] M. Wjcikowski, R. aglewski, and B. Pankiewicz, “Fpga-based real-time implementation of detection algorithm for automatic traffic surveillance sensor network,” *Journal of Signal Processing Systems*, vol. 68, pp. 1–18, 2012.

[8] K. Ratnayake and A. Amer, “Embedded architecture for noise-adaptive video object detection using parameter-compressed background modeling,” *Journal of Real-Time Image Processing*, pp. 1–18, 2014.

[9] H. Tabkhi, R. Bushey, and G. Schirner, “Algorithm and architecture co-design of mixture of gaussian (mog) background subtraction for embedded vision,” in *IEEE Asilomar Conference on Signals, Systems and Computers*, Nov 2013, pp. 1815–1820.

[10] C. Stauffer and W. E. L. Grimson, “Adaptive background mixture models for real-time tracking,” in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2, 1999, pp. –252 Vol. 2.

[11] S.-C. S. Cheung and et al., “Robust background subtraction with foreground validation for urban traffic video,” *EURASIP J. Appl. Signal Process.*, pp. 2330–2340, Jan. 2005.

[12] V. Rajagopalan, “Xilinx zynq-7000 epp: An extensible processing platform family,” *Xilinx Technical Report*, 2011.

[13] N. Farahini, A. Hemani, H. Sohofi, S. M. A. H. Jafri, M. A. Tajammul, and K. Paul, “Parallel distributed scalable runtime address generation scheme for a coarse grain reconfigurable computation and storage fabric,” *Elsevier journal of Microprocessors and Microsystems - Embedded Hardware Design*, vol. 38, no. 8, pp. 788–802, 2014.

[14] A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski, *System Design: A Practical Guide with SpecC*, 2001.

[15] Xilinx, “Zynq-7000 all programmable soc,” in *Technical Reference Manual*, 2013.

[16] Z. Wang and et al., “Image quality assessment: from error visibility to structural similarity,” *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[17] H. Tabkhi, M. Sabbagh, and G. Schirner, “Power-efficient real-time solution for adaptive vision algorithms,” *IET Computers Digital Techniques*, vol. 9, no. 1, pp. 16–26, 2015.