An Embedded Vision Services Framework for Heterogeneous Accelerators

Eduardo Gudis, Pullan Lu, David Berends, Kevin Kaighn, Gooitzen van der Wal, Gregory Buchanan, Sek Chai, Michael Piacentino
SRI International
201 Washington Rd, Princeton NJ 08543
{first.last}@sri.com

Abstract

This paper describes an architecture framework using heterogeneous hardware accelerators for embedded vision applications. This approach leverages the recent single-chip heterogeneous FPGAs that combine powerful multicore processors with extensive programmable gate array fabric on the same die. We present a framework using an extensive library of pipelined real time vision hardware accelerators and a service-based software architecture. This field-proven system design approach provides embedded vision developers with a powerful software abstraction layer for rapidly and efficiently integrating any of hardware accelerators for applications such as image stabilization, moving target indication, contrast normalization enhancement, and others. The framework allows the service-based software to take advantage of the hardware acceleration blocks available and perform the remainder of the processing in software. As performance requirements increase, more hardware acceleration can be added to the FPGA fabric, thus offloading the main processor.

1. Introduction

It has been long recognized within the embedded vision community that heterogeneous computing systems present the best solution to address the processing needs for applications powered by computer vision algorithms. There is a considerable level of data and pipeline parallelism at the pixel and feature level, combined with task level parallelism at the object classification level [2]. Companies have provided dedicated ASICs [5], as well as FPGA based processing [9][10] to solve a range of vision problems, including real-time stereo processing [6], optical flow computation[7], and object detection [8]. The challenge for heterogeneous processors, and in general, multicore solutions, is the difficulty in programming the separate cores or accelerators to best match the algorithm flow, to meet memory requirements, and to choreograph the movement of data with computation, in order to meet the desired efficiency for an embedded vision system.

This paper introduces a new embedded vision services framework that incorporates a number of dedicated hardware acceleration modules along with a service-based software architecture. The framework takes full advantage of new reconfigurable SoCs (System-on-Chip) with hardened processor cores and reconfigurable fabric. Our framework is highly adaptable, enabling existing hardware modules to be upgraded, with a "plug-and-play" capability in the provided software architecture. We have successfully implemented this framework on a Xilinx Zynq® FPGA [11], running our software on its dual ARM A9 processors and our dedicated hardware on its logic fabric.

1.1. Heterogeneous Hardware Accelerators

In this paper, we present a portfolio of specialized hardware accelerators, each implementing a particular computer vision function. These modules are optimized building blocks connected together by the user application at run time, thus forming a processing network for one or more video streams (as multiple parallel networks may also be formed). A video DMA engine with a built-in synchronization mechanism reads and writes multiple streams of data to external memory, while special purpose modules interface with multiple cameras and output displays.

The hardware accelerators are derived from the Acadia® Real Time Vision processor [3,4], which includes modules such as a three channel image fusion, local area-based image contrast normalization, image warpers for alignment, image stabilization, object detection for panning cameras, and generation of live video mosaics. The hardware accelerators operate on a multi-resolution or pyramid representation of the image of different resolutions generated by smoothing and subsampling of the image using wavelet basis functions [1]. At the lower resolution, textural information has been removed from the image such that analysis can be performed on coarse salient features of the object being analyzed. Processing at different resolution scales reduces processing workload and enhances the robustness of many algorithms, by analyzing coarse data first at low resolution, and then refining the analysis at higher resolutions. This minimizes the search space and localized correlation errors if analysis were performed only at high

978-0-7695-4990-3/13 $26.00 © 2013 IEEE
DOI 10.1109/CVPRW.2013.90

resolution.

1.2. Software Framework

The hardware accelerators are managed via constructs called video devices, each of which implements a specific defined function. All video devices use the same programming interface for status and control; additional interfaces may be added for device specific functions.

Video devices can be connected to create more complex functions, referred to as a video network. Typically one video device in the network is designated to signal the software when its video network completes its function. Video networks are typically established at initialization; their acceleration is based upon three simple interactions with the software:

- The Starting of the video network
- The Waiting for the video network to complete
- The Delivering of the result (e.g. a processed image) to another video network or software function for further processing

As programming video devices directly can be rather cumbersome, we developed a Vision Service Framework to shield the device programming details from the applications developer via vision services, a set of C++ classes. Existing Vision Services classes include many commonly used imaging functions such as stabilization and target tracking. Developers may also add their own Vision Services based upon standard templates.

2. Framework Overview

2.1. Vision IP Hardware Architecture

The hardware component of the adaptable computer vision framework consists of a collection of video devices interconnected via a crosspoint switch (Figure 1). VDMA devices read and write video frames from/to external memory into video streams, while VIN devices interface with external cameras and can accept multiple video formats. VOUT devices interface with external display devices such as HDMI/DVI. Core processing modules (VIP) connect to a comprehensive crosspoint switch, allowing the creation of multi-device networks and multiple networks running in parallel. This level of parallelism thus enables complex high performance computer vision algorithms with very low latencies.

One key feature of this architecture is the unified memory, which allows vision acceleration devices to share memory with the ARM® processor core. This is achieved by using the High-Performance port of the Xilinx Zynq Processor System to share its memory with the programmable logic component of the FPGA. This shared memory structure enables high-level vision algorithms that are better executed by the ARM than by the logic fabric,

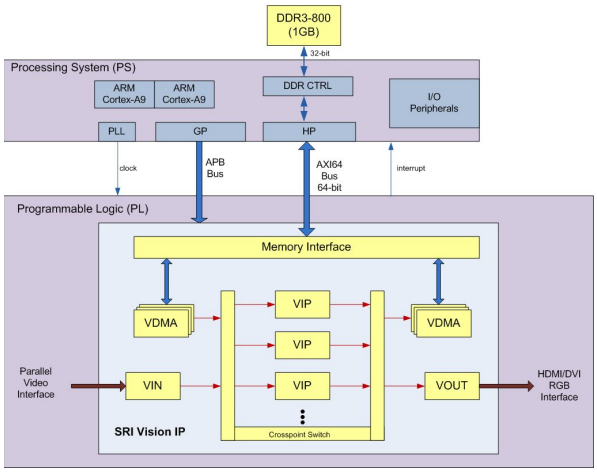

Figure 1: Vision IP Hardware Architecture

but which need the results of previous calculations performed by the hardware acceleration.

2.2. Vision Service Framework

The Vision Service Framework is a middleware that encapsulates the complexity of programming hardware devices, provides common functionalities, allows custom implementations, and implements a steady application programming interface. It is developed using the following priorities:

- Rapid development of accelerated applications
- Secure reusability, portability, and expandability
- Flexible customization

Figure 2 shows its functional block diagram.

The Vision Service Framework currently supports up to 128 video devices. Each instantiated video device occupies a slot of thirty-two 32-bit addressable registers. Each video device is specified using a text file to describe which FPGA slots have instantiated devices and of what type. Video device drivers are created accordingly for all specified video devices.

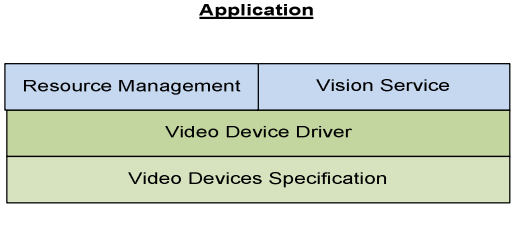

Figure 2 Vision Service Framework

Video device drivers are implemented as C++ classes deriving from the base class VideoDevice. This is an abstract class defining the common API functions that all video device drivers must implement. Figure 3 shows the

class hierarchy of some existing device drivers.

Except to support new video devices, the video device driver code is rarely changed. If Linux is used, a kernel module is needed to create device nodes for available video devices.

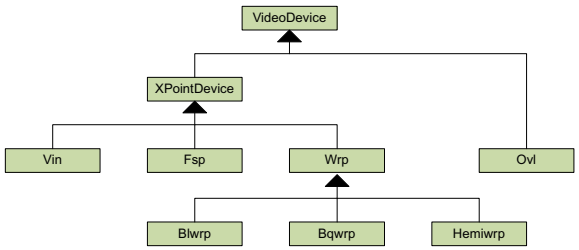

Figure 3 Video Device Driver Hierarchy

2.3. Vision Service Implementation

A vision service is a C++ class that implements a defined image-process function. Depending on its function, a vision service may use video devices for acceleration. Most built-in vision services are accelerated. A fundamental difference between a video device and a vision service is that the I/O of a video device is image pixels on the wire and the I/O of a vision service is an image buffer in memory.

The main objective of a vision service is to provide reusable building blocks for rapid development of image applications. Vision services are parameterized to support different requirements. For example, without changing code, a vision service can support different image sizes and formats by setting its parameters properly. Parameters may be set via XML configuration files at initialization or API functions at run-time.

As shown in Figure 4, all vision services are derived from the BaseService and must implement the common set of API functions defined therein. A vision service can enclose other vision services. For example the StabService in Figure 4 is only a container class providing Stabilization interfaces; the actual work is done by PryService (generates image pyramids), MeService (estimate motion), and WarpService (warps image). In a tracker application, the same PyrService and MeService may be contained under a TrackService. Custom services such as the XyzService can be added to address special needs.

Most image applications use significant memory to store images. Utility functions are provided to create pools of image buffers at initialization. All available devices are initially kept in the video device pool. Utility functions are provided for vision services to obtain and release video devices. For applications that have dedicated video devices for all functions, vision services can be set to keep the obtained video devices for the entire time. Some applications share video devices between functions. In this case vision services can be set to obtain and release device when needed.

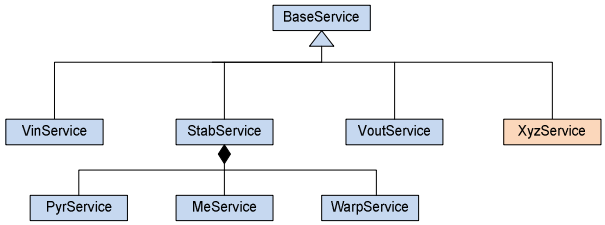

Figure 4 Built-in Vision Services

2.4. Application Development

Theoretically writing an application with Embedded Vision Service Framework is like building Legos®: one first decides what to build, then one chooses the appropriate building blocks (vision services), and finally, one puts them together. We use the MyApplication example in Figure 5 to describe some key characteristics of the Vision Service Framework.

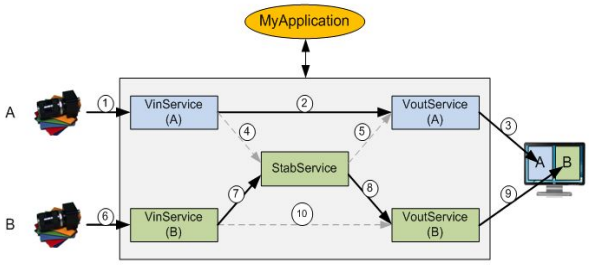

Figure 5 MyApplication Example

- What to build:

MyApplication has two video inputs, A and B. The output of A is rendered to the left half of the display; and the output of B is on the right. At any given time, one of the channels should stabilize the video.

- Choose the building blocks:

For each channel, choose a VinService to receive and correct input frames and a VoutService to render output to designated display area. Only one instance of StabService is used for the channel requiring stabilization.

- Put them together:

Write MyApplication to setup the video path A as (1,2,3) to show pass through A; and the video path B as (6,7,8,9) to show stabilized B. Upon user's request, reconstruct at run-time the video path A as (1,4,5,3) to show stabilized A; and the video path B as (6,10,9) to show pass through B.

Using Embedded Vision Service Framework, MyApplication achieved the following:

- Rapid Development: VinService, VoutService, and StabService are included in the framework. The only

new code is MyApplication.

- Deterministic low latency: each channel has its own dedicated video devices running independently. Latency is low and constant.
- Reusability: all vision services may be reused in other applications.
- Expandability: additional functions may be added in the pipeline by inserting new vision services to individual video path.

2.5. Achieving Low Latency in Linux

Since video device networks run independently, the key to minimizing latency in a pipelined processing architecture is to start each video network as soon as its input is ready. Recall that since the I/O of a video device is pixels, the input to the next video network can be ready before current video network completes. With careful timing design, a video network can be programmed to generate an interrupt after a certain number of lines are processed instead of after the last line is processed. In either case, the sooner the software that starts next video network gets started, the shorter the latency will be.

Linux divides run-time environment into user and kernel spaces. Video device drivers shown in Figure 3 are in the user space. A kernel module is required for Linux to create device nodes, handle device interrupts, and wakeup waiting threads as shown in Figure 6.

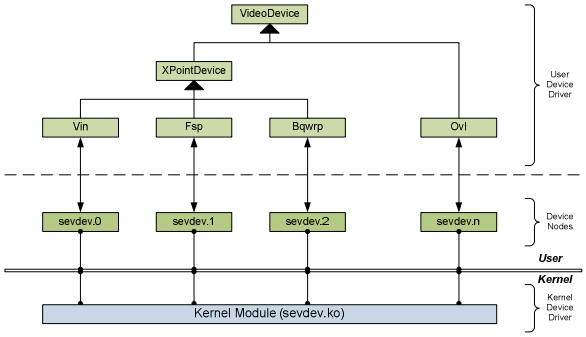

Figure 6 Linux Kernel Module

When an interrupt occurs, Linux suspends current running process and gives control to the ISR (Interrupt Service Routine), then the pending kernel threads, and at last the pending user threads.

Vision services run in the user space. The delay between an interrupt occurs and the waiting vision service gets executed varies upon CPU utilization and application complexity. For applications with stringent latency, the kernel module provides hooks for vision services to install functions to execute in the scope of IRQ and/or kernel threads.

3. Applications

We briefly describe several computer vision applications that have been ported to the framework. Each application is a real time, low latency computer vision process of a relatively complex nature.

3.1. Video Stabilization (STAB)

Video stabilization is one of the fundamental preprocessing steps in image processing. It is used to align successive video frames captured from a moving or shaking platform, or to match two or more images of the same scene taken at different times, from different viewpoints, and/or from different sensors. This process is based on the registration, or alignment, of matching pixels between video frames. The SRI patented pyramid-based video registration application provides highly accurate video registration and stabilization; its algorithm consists of pyramid generation, image correlation and motion estimation. The result of the frame-to-frame motion is used in a warping function to render the display. Figure 7 shows an example for pre and post image stabilization.

Figure 7 Stabilization Example

The video frames to be stabilized are fed into the registration process. The pyramid generation consists of multi-scale multi-level Laplacian images. Each Laplacian image contains contrast features of the scaled original image. The correlation and motion estimation block computes the motion or 2-D displacement between the incoming frame I(t) and a reference frame from an earlier time I(t-Δt). The motion estimation algorithm estimates the motion successively over a dozen iterations, each time yielding a more precise answer, until the registration of the pixels reaches the accuracy of up to a tenth of a pixel.

Many different applications beyond basic stabilization are enabled through the use of image registration. Once frames are aligned to a fraction of a pixel, they can be further manipulated to support other vision plug-ins, including video enhancement, mosaic construction, video fusion, change and motion detection, image compression, and 3D depth mapping from stereo imagery.

3.2. Moving Target Indication (MTI)

The detection of moving objects from a fixed camera is performed by simple arithmetic operations and filtering of the video image. The simplest method for detecting independently moving objects is to subtract one frame from a previous frame. The observed change indicates where the objects have moved in the scene. However, due to factors such as noise and changes in lighting, some filtering is required to extract only the movers and avoid spurious detections. The algorithm implemented for the

pyramid-based MTI application uses a Laplacian representation for the images and computes change between the adjacent frames.

The change image is binarized by setting a threshold on the difference term at each pixel, which is computed in a 5x5 window at each pixel in a Laplacian image. This binary image is a convenient representation for the "blob tracker." Once independently moving objects have been detected in the scene, they can be tracked over time.

Blob tracking, so named because it uses input images that have binary change mask (blobs), is done as follows:

- Compute the change detection between the adjacent frames.
- Perform connected components analysis on the detected change regions.
- Extract blobs from the connect components.
- Merge blobs to remove spurious blobs belonging to a single moving object.

Additionally a tripwire detection feature, allows the user to define an arbitrarily positioned "tripwire" across the camera field of view. Whenever a tracked object crosses or intersects with the tripwire an alarm event is triggered.

3.3. Contrast Normalization (CN)

The Contrast Normalization application provides dramatic dynamic rage enhancement of video images as shown in Figure 8. Advanced imaging sensors used by the military often have a dynamic range that far exceeds that of displays used to present these images to human observers. The Contrast Normalization method can be used for compressing the overall dynamic range of such images while enhancing the visibility of salient pattern information. The method operates in a multiresolution Laplacian pyramid (or wavelet) domain. It has the effect of adjusting the contrast of individual features within the image in order to optimize their visibility. As such it differs from methods commonly used today, such as window and scale or histogram normalization, which operate directly on pixel intensity values without regard for image pattern structure.

Contrast normalization is motivated by the observation that there is an optimal contrast for the visibility of features in an image. When contrast is significantly below this optimum, features become difficult to see, or can be masked by nearby higher contrast features. Contrast normalization seeks to adjust the contrast of features in a source image towards the optimal. Features that are below optimal are increased in contrast. Features that are above the optimal are decreased in contrast.

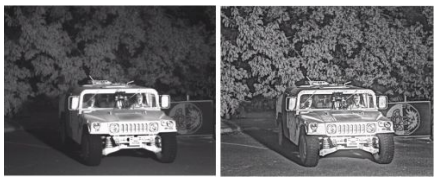

Figure 8 Contrast Normalization Example

4. Results

The three applications described in section 3 were implemented on two Xilinx Zynq platforms: the Xilinx ZC702 evaluation board (Figure 9) that has the 7020-1 part, and a custom board that has the 7045-1 board. The applications running on the ZC702 board were monochrome because of the limited resources available. Using the 7045 larger FPGA part, we were able to run both Stabilization and Contrast Normalization simultaneously at 60Hz as well as numerous other image manipulation functions.

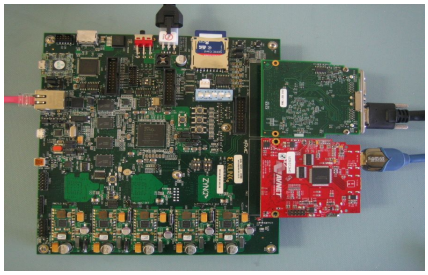

Figure 9 ZC702 Evaluation Board

The ZC702 FPGA video processing clock rate was 100MHz at the slow speed grade. That system used a single bank of DDR3-1066 shared between the video acceleration hardware modules and the ARM processor. The 7045 video processing clock rate was 142MHz using the slow speed grade part. The 7045 system used a dual memory configuration, a DDR3-1066 for the ARM processor and a DDR3-1600 for the hardware acceleration modules allowing much higher performance. The ARM processor was running at 800MHz on both platforms.

TABLE 1: APPLICATION IMAGE RESOLUTION AND FRAME RATE

Platform	Application	Image Resolution	Frame Rate
ZC702	**STAB**	640 x 480	30
ZC702	**MTI**	1280 x 1024	15
ZC702	**CN**	1280 x 1024	30
7045	**STAB + CN**	1280 x 1024	60

The applications implemented used a variety of hardware acceleration modules from the Acadia Vision IP portfolio plus the service based framework running Linux on the ARM. It also included a camera input and an HDMI display output. Table 1 describes the image resolution and frame rate of various applications.

Table 2 shows the resource utilization required to run the different applications. Note that the resources for each application are highly dependent on the desired performance, frame rate and video resolution. The STAB resources are also highly dependent on the type of camera movement and distortion correction.

TABLE 2: RESOURCE UTILIZATION

APP	ARM	FFs	LUTS	DSP	BRAM
STAB	<25%	25K-33K	20K-32K	80-140	50-140
MTI	<35%	11K-13K	10K-12K	30-45	15-30
CN	< 5%	25K-32K	22K-27K	40-50	40-90

The STAB application can be configured to use from 38% to 60% of the LUTs, and from 19% to 50% of the BRAM of the Xilinx 7020 Zynq FPGA. The range of utilization values represents the configurability of the hardware accelerator to meet the different requirements of the application (e.g. type of motion, resolution, etc.). The MTI relies mostly on the ARM, using from 19% to 22% of the LUTs. The CN application uses very little ARM processing, relying mostly on the hardware acceleration; it uses from 41% to 51% of LUTs on the 7020 FPGA. All three applications could be combined to run simultaneously at maximum performance on the larger 7045 FPGA, and would use up to 60% of its ARM, 32% of its LUTs and 23% of its BRAM, thus leaving significant resources available for other applications. The resource utilization is much lower for lower resolution cameras or translation/scale only stabilization.

The framework developed for this heterogeneous system is adaptable to the unique requirements of the desired system. A smaller FPGA can be used when a low power, low cost, single application is required. The framework allows the service-based software to take advantage of the hardware acceleration blocks available and perform the rest of the processing in the ARM. As performance requirements increase, more hardware acceleration can be added to the FPGA fabric, thus offloading the ARM processor. The framework also allows multiple applications to run simultaneously, taking advantage of the parallel nature of the hardware/software architecture. In the case of high resolution high frame rate processing (e.g. 1080p60 HDTV), the framework can be adapted for a dual external memory structure by populating the FPGA and the service-based firmware with the appropriate modules.

5. Conclusion

This paper presented an adaptable computer vision framework for heterogeneous FPGA architectures consisting of hardware-based Vision IP modules and a service-based software architecture running on Linux. The hardware processing component is built from a library of various hardware modules interconnected by a configurable crosspoint switch and a DMA engine that sends and receives synchronized video to and from memory. The software piece run on the ARM processor core and is composed of device drivers and vision services, which are stitched together to form vision applications.

We described three applications built under this framework: stabilization, moving target indication, and contrast normalization. They were implemented on a low power, low cost, high-performance platform that clearly demonstrated the adaptability of the framework. This framework, however, is not limited only to these three applications in computer vision. We plan to develop many more applications running on heterogeneous FPGAs including from stereo vision, high dynamic range, feature tracking, object recognition, visual navigation, and augmented reality among many others.

References

[1] P.J. Burt and E. H. Adelson. The Laplacian pyramid as a compact image code. Trans. Comm., 31(3):532-540, 1983.

[2] A.Lopez-Lagunas, S. Chai, “Streaming Data Movement for Real-Time Image Analysis,” *Journal of Signal Processing Systems*, vol. 62, no. 1, pp.29-42

[3] G. van der Wal, “Technical Overview of the Sarnoff Acadia II Vision Processor,” SPIE Defense, Security, and Sensing Conference, Subconference 7710: Multisensor, Multisource Information Fusion: Arch., Algs, & Apps, Orlando, April 2010, Proc. SPIE 7710, (2010).

[4] G. van der Wal, et. al., “Acadia II: A Heterogeneous Many-Core SoC for Multi-Resolution Embedded Vision Processing Systems. Workshop on SoC Architecture, Accelerators & Workloads, San Antonio, Texas, Feb 2011.

[5] Dagan, Erez, et al. "Forward collision warning with a single camera." *Intelligent Vehicles Symposium, 2004 IEEE*. IEEE, 2004.

[6] Isakova, Nilufar, S. Basak, and A. C. Sonmez. "FPGA design and implementation of a real-time stereo vision system." *Innovations in Intelligent Systems and Applications (INISTA), 2012 Intl Symp on*. IEEE, 2012.

[7] Gultekin, Gokhan Koray, and Afsar Saranli. "An FPGA Based High Performance Optical Flow Hardware Design for Computer Vision Applications." *Microprocessors and Microsystems* (2013).

[8] Brousseau, Braiden, and Jonathan Rose. "An energy-efficient, fast FPGA hardware architecture for OpenCV-Compatible object detection." *Field-Programmable Technology (FPT), 2012 Intl Conf on*. IEEE, 2012.

[9] Benkrid, Khaled, Danny Crookes, and Abdsamad Benkrid. "Towards a general framework for FPGA based image processing using hardware skeletons." *Parallel Computing* 28.7 (2002): 1141-1154.

[10] Lim, Yoong Kang, Lindsay Kleeman, and Tom Drummond. "Algorithmic methodologies for FPGA-based vision." *Machine Vision and Applications* (2012): 1-15.

[11] Xilinx, Inc., “Zynq-7000 Extensible Processing Platform,” http://www.xilinx.com/zynq