# IMAGE PROCESSING FRAMEWORK FOR FPGAS

## *Introducing a Plug-and-play Computer Vision Framework for Fast Integration of Algorithms in Reconfigurable Hardware*

Bennet Fischer and Raul Rojas

*Intelligent Systems and Robotics, Free University of Berlin, Arnimallee 7, Berlin, Germany*

Keywords: Image Processing, FPGA, Framework, Signal Processing, Embedded.

Abstract: This paper presents a framework for computer vision tasks on Field Programmable Gate Arrays (FPGA) which allows rapid integration of vision algorithms by separating the framework from the vision algorithms. A vision system can be created by using plug-and-play methodology. On an abstract level several input and output channels of the system can be defined. Also, commonly used image transformations are modularized and can be added to the inputs or outputs of an algorithm. Special input and output modules allow the integration of algorithms with no knowledge of the surrounding framework.

## 1 INTRODUCTION

Two dimensional signal processing has ever since been a demanding problem in computer science. Computer vision as an application of the latter is still a very active research area. With the steady improvement of general computer hardware, many computer vision algorithms are able to run in real time on commodity hardware. The importance of real time implementations is growing as computer vision is starting to be employed in every day products like game consoles, mobile phones and driver assistance systems. Most of these systems are by their nature embedded systems. However, many of the fundamental algorithms i.e. dense 3D reconstruction or optical flow estimation are still not applicable to real time implementations. This applies to commodity pc hardware and in particular to embedded processors.

Recently programmable graphics hardware gained popularity in research for realizing real time implementations of demanding vision algorithms. This type of hardware can be programmed in a familiar way using the high level "C" language and thus requires only a short training period for new users. The level of hardware abstraction is comparable to CPU programming. Memory access, input/output operations are all mapped to easy to use operations. This makes graphics hardware attractive for quickly evaluating algorithms. However, due to their high power consumption they are not well suited to embedded systems.

An established method of computational intensive real time implementations is the use of field programmable gate arrays (FPGA) devices. FPGAs generally suite better to the needs of embedded systems due to their lower power consumption (Jin et al., 2009). This makes them attractive for appliances outside of the researchers laboratories i.e. in autonomous systems. A detailed comparison of general purpose processors (GPP), graphic processors (GPU) and FPGAs is given in (Cope et al., 2009). The main disadvantage of FPGA systems is, however, the high development effort. New users are faced with a steep learning curve. The time to the first productive use of the device is usually long for two reasons:

- The user has to learn a new programming language which describes hardware, not software. Also the tool chain for implementing the written programs is completely different to software tool chains. High level synthesis tools trying to bring the FPGA closer to the programming model of traditional processors can help new users to develop algorithms faster than before (BDTI, 2010). However, these tools are in most cases not affordable for researchers or small companies.
- The hardware abstraction on FPGAs is poor, if not absent. In the case of no abstraction, the peripheral hardware is simply wired to the input/output (io) banks of the FPGA. All the higher levels of abstraction have to be done by the user. One common concept of abstraction in the design of hard-

Fischer B. and Rojas R..
IMAGE PROCESSING FRAMEWORK FOR FPGAS - Introducing a Plug-and-play Computer Vision Framework for Fast Integration of Algorithms in Reconfigurable Hardware.
DOI: 10.5220/0003801402950300
In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems* (PECCS-2012), pages 295-300
ISBN: 978-989-8565-00-6
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

ware is the use of intellectual property (IP) cores. Theses cores cover the underlying complexity of hardware peripherals and offer a more abstract interface to the functionality. Most FPGA vendors offer IP cores for often used peripherals like Random Access Memory (RAM) or networking hardware. However, the interface to these cores is still complex as they usually offer bus interfaces (PLB, AXI, Whishbone, etc.). To access these kind of cores, the user has to implement a bus participant, thus needs to know the bus specification in detail. This is a non trivial task and highly time consuming.

In this paper we present a framework aimed at accelerating the development of FPGA vision algorithms. The problems of FPGA development stated above, especial the latter one, are overcome by a high level of peripheral hardware abstraction. This abstraction specifically suits to the needs of computer vision algorithms.

The typical use case of this framework is the creation of a real time capable prototype system. In contrast to GPU based real time prototypes, the implementation of the algorithms is much closer to a production ready state. This allows the user to predict the overall cost and energy consumption of the system very precisely. It is assumed that the algorithms are already evaluated using a non real time capable implementation. This reference implementation can then be ported to the FPGA and be deployed with little effort. The user can concentrate on developing the vision algorithms and is not forced to invest time in retrieving and passing on the data.

The focus here is not to provide an overall high level of abstraction covering also the vision algorithm itself. Instead, the framework allows to test and use production ready HDL implementations of algorithms without any infrastructural developing overhead.

## 2 RELATED WORK

A FPGA co-processor framework is presented in (Kalomiros and Lygouras, 2008). Several vision algorithms are evaluated using a commercial *Simulink* to HDL translator. Communication to the host PC is done via USB and the data flow is organized by a soft processor. As being a co-processing system with no direct access to the image data, the latency is higher than in a pre-processing system but the possible range of applications is broader.

A framework for verification of vision algorithms is presented in (van der Wal et al., 2006). Conceptual similar to the approach described by us, they use image pipelines to process the data. However, due to the crosspoint switch, their processing entities can be connected at run-time, allowing high flexibility. This flexibility is useful for hardwired Application Specific Integrated Circuits (ASIC) which cannot be reconfigured.

## 3 FUNDAMENTAL CONCEPTS

The framework itself consists of *modules* connected by *streams* and a *supervisor* organizing the system configuration and data flow. It is assumed that the platform on which the framework is running consists of at least an FPGA, an external RAM and a communication module to a workstation PC (gigabit ethernet, PCI Express, etc.) all interconnected by the system bus.

A *module* encapsulates an arbitrary function. In the simplest form a module operates on data of the input streams and delivers the result on one or several output streams. More complex modules also interact, besides the streams, with lower level components like bus interfaces or hardware peripherals. However, to the user this complexity is opaque as only the stream interfaces are visible to him. Modules can be instantiated and connected as hardware description language (HDL) entities in source files or more convenient to the user, via a graphical user interface (GUI) by dragging them into the system to be built.

A *stream* is a unidirectional data flow interface. The most common use of it is to transfer pixel values. Note that this interface is kept as simple as possible. The synchronization of this interface is only word-wise. Every other synchronization information has to be implicit which means that the data format of a stream has to be known a priori. This is in general true for image processing algorithms. The implicit synchronization offers several advantages. First, due to the fixed input format, the module can process the data in a statically way. This eases the development and normally speeds up the implementation. Second, the module can trust the format of the input data. No error checking has to be done on the format of the input data. This leads to a better encapsulation of functionality as an exceptional state will be handled inside a module instead of being passed between two modules. One prerequisite of this to work is that every module delivers correctly synchronized data on its output streams. In most cases this is easier to archive than format error checking on the input streams in case of explicit synchronization.

The *supervisor* is a general purpose soft processor with low speed requirements. It organizes the data flow from the FPGA system to the host application on a data packet base. As the data packages are already prepared by hardware components, the load of this processor is low. Another task for the supervisor is to initialize and configure the hardware modules. For normal applications the user does not need to change the software running on the processor. Drivers are bound to specific modules and, if necessary, inserted into the software automatically.

## 4 CORE MODULES

As stated above, the basic building block of the framework is a module. The modules are grouped into three categories:

- IO modules.
- Processing modules.
- Simulation/Debug modules.

### 4.1 IO Modules

#### 4.1.1 Live Data Source

The live data source (LDSO) module offers a single output stream from a hardware device. Typically it delivers a pixel stream from a camera sensor.

#### 4.1.2 Memory Data Source

The memory data source (MDSO) module takes data from an external RAM device and transforms the data to one or several output streams. It is connected to the RAM controller via the system bus and to the supervisor via a control interface. The bus interactions are fully encapsulated by the module. By using this module instead of the live data source as the algorithm input, the system transforms from a pre-processor to a co-processor. Image data can be delivered by the workstation PC, processed by the system and sent back.

#### 4.1.3 Synchronized Data Source

The synchronized data source (SDSO) takes up to three equally formatted input streams and synchronizes them to pixel accuracy. The synchronized data is then sent on three output streams. In case of no possible synchronization (due to buffer size restrictions) the output channels are fed with pixels marked as invalid. This module is useful to transform loosely synchronized camera sensor input to pixel synchronous streams.

#### 4.1.4 Memory Data Sink

The memory data sink (MDSI) is the counterpart to the memory data source. To the user it provides a single stream input, to the system a bus connection and to the supervisor a control interface. The user can send data to the module with a stream which will be orderly written into the external RAM. The data is packetized, addressed and sent via the system bus into the frame buffers in RAM. Also, the supervisor is informed when new data has arrived in memory. This information can be used by the supervisor to transfer the data to a workstation PC via gigabit ethernet or pci express. Already available scatter-gather direct memory access (DMA) components allow the supervisor to transfer the data with zero copy overhead.

### 4.2 Processing Modules

In the following section two examples of processing modules are given. These modules are part of the framework and can be used to compose more complex functionalities out of them. Two examples of their usage will be showed later on.

#### 4.2.1 Separable Convolution

The separable convolution (SC) realizes a 2D finite response filter. Using this module many standard operations can be performed by adapting the filter coefficients. The coefficients can be configured at build time.

#### 4.2.2 Geometric Image Transformation

The geometric image transformation (GIT) module takes a pixel stream and performs an arbitrary geometric transformation on the image. The transformation function can be changed in the running system. This function is usually used to remove lens distortion or rectify sets of image streams.

### 4.3 Simulation/Debug Modules

#### 4.3.1 Memory Data Source

The memory data source module can be re-purposed as a debug module. By replacing a live data source with it, the system can process artificial or static images.

#### 4.3.2 File Data Source

The file data source (FDSO) is a non synthesizable simulation module. It can be used as a test bench during simulation to verify the functionality of a module. The FDSO reads data from image files or comma separated files and transforms the data to a stream. This stream can be connected to a module under test as the input stimuli.

#### 4.3.3 File Data Sink

As with the MDSO, the file data sink (FDSI) is the counterpart of the FDSO. In a simulation the output stream of a module under test can be connected to the FDSI. The data received is written orderly into a file, making it available for post-simulation verification i.e. the check against a reference implementation. The module verification is visualized in figure 1.

## 5 EXAMPLE APPLICATIONS

The presented framework has been in use since one year at the Free University of Berlin inside the autonomous car "Made in Germany".

Automotive image processing requires low latency as well as low power consumption. These requirements make the use of FPGA hardware attractive.

The image data is delivered by two CMOS cameras mounted behind the windshield with 768x500 high dynamic range (HDR) images at 30 frames per second. The processed data is sent to a laptop via gigabit ethernet for higher level processing. This setup frees the laptop from the highly time consuming low level vision algorithms.

The data flow graphs illustrating the examples consist of modules visualized as boxes and streams visualized as arrows.

### 5.1 Module Verification

Before being uploaded to the hardware, modules need to be verified. The framework supports verification through the FDSO and FDSI modules. A basic verification setup is illustrated in figure 1. Both the reference implementation and the hardware implementation getting the same stimuli and write their results to a comma separated file. If the hardware implementation is behaviorally correct, the two result files should be identical.

### 5.2 Optical Flow

The first application example for the framework is the estimation of the optical flow (Lucas and Kanade, 1981).

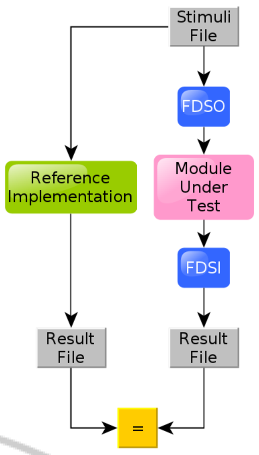

Figure 1: Module verification.

What follows is a brief overview of the Lucas-Kanade algorithm:
As a first step, the spatial derivatives are calculated.

$$I_x(x,y) = \frac{I(x+1,y) - I(x-1,y)}{2}$$
$$I_y(x,y) = \frac{I(x,y+1) - I(x,y-1)}{2}$$

Next, the Spatial gradient matrix $G$ is determined where $w$ describes the size of an integration window. This matrix is also called structure tensor.

$$G = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} I_x^2(x,y) & I_x(x,y)I_y(x,y) \\ I_x(x,y)I_y(x,y) & I_y^2(x,y) \end{bmatrix}$$

By using the temporal derivative $\delta I$ and the spatial derivatives $I_x, I_y$ the image mismatch vector $b$ is calculated.

$$\delta I(x,y) = I(x,y) - J(x,y)$$
$$b = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \delta I(x,y)I_x(x,y) \\ \delta I(x,y)I_y(x,y) \end{bmatrix}$$

The following equation then gives the estimate of the optical flow $\eta$:

$$\eta = G^{-1}b$$

The matrix $G^{-1}$ is also known as the covariance matrix. The algorithm is explained in detail in (Bouguet, 1999).

As a prefiltering step the image pyramid is created and written into the external RAM. The module structure of the prefilter is shown in figure 2.

After the arrival of a frame from the prefilter, the supervisor triggers the MDSO module seen in figure 3. The current frame $I$ and the preceding frame $J$ are

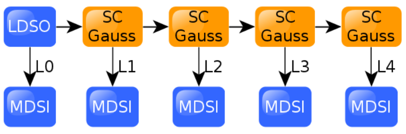

Figure 2: Preprocessing modules forming Gaussian pyramid: A live image is low pass filtered four times by a Gaussian kernel (SC). Each filter result is written to RAM (MDSI).

simultaneously input into the optical flow estimator. Note that the estimator has no other dependency to the framework than its stream interfaces.

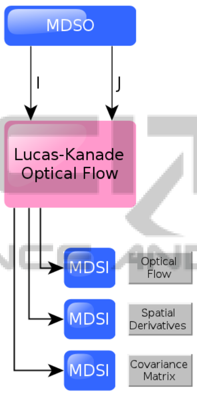

Figure 3: Lucas Kanade optical flow integration.

The visualized result of the optical flow estimator can be seen in figure 4. The colors denote the direction and the intensity the speed of the flow. Reference colors are found on the border of the image.

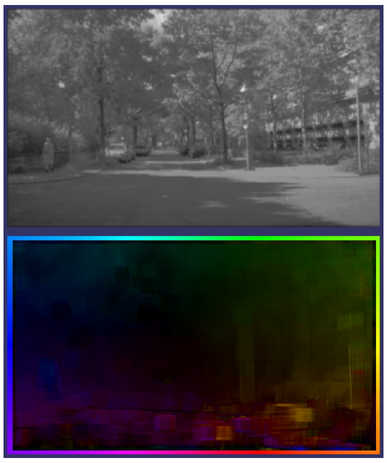

Figure 4: Top: Source image. Bottom: Optical flow result.

## 5.3 Stereo Vision

The second example for the framework is a module for estimating the distance of objects by finding corresponding image points in a stereo image pair. As shown in figure 5 the geometric image transformation (GIT) module is used as a preprocessing step to remove the lens distortion and rectify the images. The transformed streams are then synchronized to pixel accuracy by the SDSO module.
What follows is a brief description of the block matching algorithm:

The cost $C$ for the comparison of two blocks of size $w$ at the disparity $d$ is defined by the sum of squared differences.

$$C(x,y,d)=\sum_{x=p_x-w_x}^{p_x+w_x}\sum_{y=p_y-w_y}^{p_y+w_y}(L(x,y)-R(x-d,y))^2$$

The disparity $D$ with the lowest cost over the search window $D_{max}$ is the estimate of the disparity.

$$C_{min}(x,y)=\min_{0\leq d\leq D_{max}} C(x,y,d)=C(x,y,D)$$

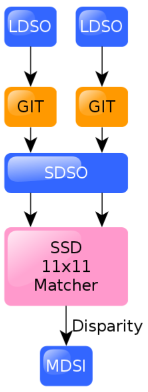

Figure 5: Stereo vision integration: Two live images (LDSO) being rectified (GIT) and synchronized (SDSO). The streams are block matched and the disparity is written to RAM (MSDI).

The result of the depth estimation on a street scenario is visualized in figure 6. The color denotes the disparity of the pixel. Occluded and low textured areas are filtered out with a left to right check and a confidence check regarding the uniqueness of the minimal cost. Also a sub-disparity interpolation is performed, resulting in a ready to use disparity image.

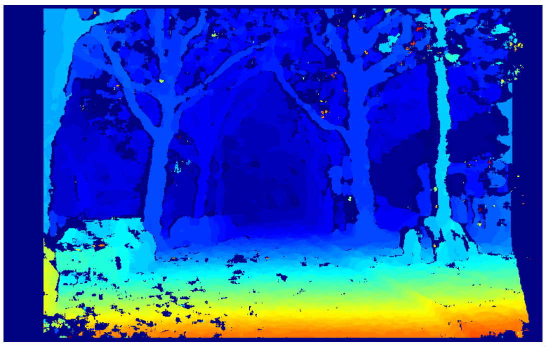
Figure 6: Stereo Vision result of a street scenario.

## 6 SUMMARY

In this paper we have presented a framework helping researchers to quickly evaluate production ready vision algorithms on FPGAs. The strict modularization of functionality and small dependencies between modules allow the user to quickly change the functionality of the whole system. Functionalities can be added or removed depending on the requirements of the application and the available chip area.

The system proofed to fulfill the goal of fast algorithm integration and reliable operation in the autonomous car "Made in Germany". Future developments will target on porting the upper level algorithms interpreting the preprocessed data to an embedded system. Examples for these algorithms are automatic camera calibration or obstacle avoidance currently running on a commodity laptop.

## ACKNOWLEDGEMENTS

We would like to thank Robert Richter for his work on implementing the Lucas-Kanade algorithm.

## REFERENCES

BDTI (2010). The AutoESL AutoPilot High-Level Synthesis Tool. *http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf*.

Bouguet, J. (1999). Pyramidal implementation of the lucas kanade feature tracker description of the algorithm. *Intel Corporation, Microprocessor Research Labs, OpenCV Documents*, 3(2):1–9.

Cope, B., Cheung, P., Luk, W., and Howes, L. (2009). Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on Computers*, 59(4):433–448.

Jin, Q., Thomas, D., and Luk, W. (2009). Exploring reconfigurable architectures for explicit finite difference option pricing models. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, volume 54, pages 73–78. IEEE.

Kalomiros, J. and Lygouras, J. (2008). Design and evaluation of a hardware/software FPGA-based system for fast image processing. *Microprocessors and Microsystems*, 32(2):95–106.

Lucas, B. and Kanade, T. (1981). An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 674–679.

van der Wal, G., Brehm, F., Piacentino, M., Marakowitz, J., Gudis, E., Sufi, A., and Montante, J. (2006). An FPGA-based verification framework for real-time vision systems. *Pattern Recognition*, 2.