A NEW HIGH-LEVEL METHODOLOGY FOR PROGRAMMING FPGA-BASED SMART CAMERA

Nicolas Roudel, François Berry, Jocelyn Sérot

LASMEA
UMR 6602 CNRS - Université Blaise Pascal
24, Avenue des Landais
63177 Aubière Cedex
France
firstname.name@univ-bpclermont.fr

Laurent Eck
CEA, LIST,
18 route du Panorama, BP6,
Fontenay-aux-roses,
F- 92265 France
firstname.name@cea.fr

ABSTRACT
Due to the various devices composing a smart camera system, various languages have to be known by the designer (like HDL and C/C++). Most of vision applications designers are software programmers and do not have a good knowledge of HDLs (VHDL). This paper presents a new high-level methodology for implementing vision applications on smart camera platforms. This methodology is based on a soft-core approach to manage the whole system and a dataflow (actor-oriented) language to design the processing elements. We discuss in particular interfacing constraints.

Keywords-Soft-core, methodology, smart camera, heterogeneous platforms, CAL, processing elements;

I. INTRODUCTION
Since the 1990s, smart cameras have grown popularity and market acceptance. The constant evolution in VLSI technologies has made smart cameras an active area of research both in industrial and academic communities. A smart camera is a vision system whose primary function is to produce a high-level understanding of the observed scene, generate application-specific data and transmit only useful informations to the host system. Thus, the classical bottleneck due to communications between the sensor and the processing units is significantly reduced and the received data can be immediately used by the host system.

Smart cameras are usually composed of an image sensor, several and various processing devices and a network processor. Nowadays with the large integration possibilities, it is common to have several processing devices, such as GPU, FPGAs or DSP, on the same platform. This kind of architectures are called heterogeneous platforms and offers a good trade-off between high performances and good flexibility. On the other hand, the design of applications within these platforms is difficult, particularly due to schedule and control aspects.

A solution to simplify these aspects is the use of a soft-core approach. With developments in FPGA logic density and speed it is now possible to implement a complete system on a programmable chip (SoPC). Systems in which one or more processor cores and modules (peripheral drivers and processing elements) written in HDL are put into one single FPGA and this leads to heterogeneous multiprocessor architectures.

In a soft-core approach, the processing is performed by the core itself. However, the system designer can extend the soft-core functionality by adding specific IPs and defining custom instructions. These IPs are usually called PEs (Processing elements) and are the basis of this paper. Various methodologies can be used to describe PEs :

- First approach is to describe them directly with a HDL language (VHDL,Verilog). But this drastically reduces the advantage of the soft-core approach namely the fact that the processor can be programmed by end-users without any HDL knowledge. As a matter of fact, there is little interest in controlling the whole system with a high-level language and designing PEs with a low-level language.

- The second approach, relies on C-like languages, like handel-c[2] or impulse-c[1]. This method has the advantage that software designers can easily program with this kind of tools. Nevertheless, the fact that parallel applications are omnipresent in image processing, strongly reduces the applicability of this approach because C-like languages are intrinsically sequential.

- Another method is the use of graphic user interfaces (GUIs), like simulink[3], IVC studio[4] or CMUcam2 GUI[5]. Although this approach has the advantage that programming becomes very intuitive, the flexibility
is strongly reduced due to the fact that each GUI is strongly bound to a given specific architecture.

This paper presents an alternative methodology. We propose to use a dataflow/actors-oriented language to describe PEs. The language used is CAL [6]. The presented work describes the integration of PEs designed with CAL in a custom soft-core, called SeeCORE.

This paper is organised as follows. Section 2 presents our custom soft-core architecture. In Section 3, methodology for designing processing elements is described. In Section 4, a comparison, based on the implementation of an optical flow computation algorithm, between our methodology and a HDL hand written version is reported. Finally, conclusion and several perspectives are given in Section 5.

II. A SOFT-CORE BASED ARCHITECTURE

To evaluate the proposed methodology, a virtual processor written in HDL language, called SeeCORE, has been developed. This soft-core is programmed using a simplified assembly language with a reduced instructions set. As shown in Fig. 1, the processor is composed of five parts:

- A control and schedule unit (CSU),
- A set of hardware managing modules, called drivers, for interfacing to hardware peripherals,
- A processing unit composed of vector arithmetic and logic unit (VALU) and a set of processing elements (PEs),
- A set of memory banks,
- A Crossbar device, allowing simultaneous use of different memory banks and their dynamical re-allocation to the various drivers/PEs.

The CSU is classically composed of five intercommunicating modules: a program memory, a program counter, a control unit, an instruction register and a register file unit. It is programmed with a custom assembly language. Instructions are composed of an 8 bit opcode and two n bit operands where n can be defined by the designer. This n value thereby defines the size for addresses, data buses and registers.

The instruction set consists of arithmetic, data and control flow instructions. To configure drivers and PEs, the LOAD instruction is used. When this instruction is decoded by the central unit, the register file updates parameters. The register file unit has three various register types:

- The write-only registers are used to set drivers/PEs parameters like image resolution, integration time or number of packets to transmit,
- The read-only registers contain receipt status or results from external devices. For example, if a PE computes an average value on an image it can directly store the result in a read-only register instead of storing this value into memory,
- The data registers are used for variables and loop indexes.

Written in HDL language, peripheral drivers are used to control, according to the CSU instructions, the external devices such as the image sensor, inertial sensors or communication devices. For a given architecture, these drivers do not have to be modified.

The processing part is composed of a vector arithmetic and logic unit, detailed in [7], and a set of processing elements. The role of and the design methodology for the PEs, which is the innovating part of this paper, are discussed in the next section.

Both the processing and peripheral drivers parts communicate through a crossbar device using a specific control protocol. The crossbar is a key element of this architecture. It allows simultaneous use of several memory devices, as well as dynamical reconfiguration of the data path without full FPGA reconfiguration. Being asynchronous, crossbar operations are independent of each connected element. Crossbar configuration is defined by a control word which indicates the memory bank assigned to each input port. Consequently, it is possible to use strategies such as memory swapping or simultaneous use of several memory devices by several processing elements.

Let’s take for example, an application composed of an acquisition, a processing and a transmission step running on a platform with three independent memory banks. With a shared bus, acquisition stores an image in memory 1 and then the bus control is given to the processing element and image sensor can not acquire a new image. In the case of distributed memories system, it is necessary to copy the content of a memory to another. Using the crossbar, a re-allocation of the memory 1 to the PE, and the memory 2 to the sensor is possible. Thus, both can continue to work as shown in Fig. 2.

In practice, simultaneous control of the PEs and drivers can be difficult since each element can work at this own clock frequency. An asynchronous control protocol is therefore used. Drivers and PEs are controlled using a set of registers (detailed above) and a set of flags implementing an asynchronous handshaking protocol. The number of flags is flexible and it is possible to add a pair of flags whenever a new PE/driver is added.

III. PROCESSING ELEMENTS DESIGN METHODOLOGY

As introduced in Section 1, the originality of proposed methodology is to rely on a dataflow/actor-oriented
III-A. CAL Language

The CAL language [6] has been created in the Ptolemy II project at Berkeley's university. A dataflow model expressed in CAL is composed of a set of independent actors and their connection structure, as a network of actors. An actor is a stand alone entity which has its own internal state and which performs computations by firing actions. It has a set of input and output ports through which it communicates with other actors by exchanging data tokens. An actor must have, at least, one fire able action to do computations. Actions execute (or fire) based on the internal state of the actor and depending on the availability and values of tokens at the input ports. An action may consume tokens from inputs, change the internal state of the actor, and produce tokens at the outputs. Action execution is modelled as an atomic component which means that no other action of the same actor can execute while an action is executing or interrupting any executing action. CAL provides scheduling concepts to control execution order of actions inside an actor. CAL actors can be combined into a network of actors to build larger systems. This is achieved by connecting the input and

---

**Fig. 1.** SeeCORE architecture.

**Fig. 2.** Example of memory swapping strategy using the Crossbar.

language for designing the PEs. This choice has been motivated by the fact that this programming model is particularly suited to the design of parallel applications and in particular to vision applications. We use the CAL dataflow language [6]. A brief account of this language is given in section 3.1. In section 3.2, we describe how PEs, described in CAL, can be inserted in our soft-core based design flow.
III-B. PEs flow integration

In our case, inserting the CAL design flow within our soft-core based design flow requires that the actors implementing the PEs meet some interface requirements. The involved transformations are sketched in Fig. 5.

First, an actor handling the asynchronous handshaking protocol has to be inserted (HS actor in Fig. 5). It takes as input the flag produced by the CSU (Flag in in Fig. 5) signalling the start of the processing and propagates it along the design through the flags management module of each actor (described below). It also takes as input the End Of Work (EOW) signal from the actor handling memories which indicates the end of processing and generates the global output flag (Flag out in Fig. 5) which is sent to the CSU.

Second, an actor handling accesses to memories also has to be inserted (MEM actor in Fig. 5). This actor retrieves data from a memory bank (for example, an image produced by the image sensor). Furthermore, it has to manage the storage of processed data and to inform the HS actor that the processing is finished.

Third, actors describing the PE functionality, through a network like in Fig.3, themselves have to be modified. For each actor of the network, intrinsic modifications have to be applied. As said above, the management of flag signals has to be included. For each actor, as long as the Flag in is low, it has to be in a sleeping state. Once high, the actor begins the processing and put the Flag out to high to enable the next actor of the network. Then, to take parameters, sent by the CSU, into account, additional input ports must be added. At last, for optimisation considerations, bus sizes are defined in each actor for each in/out port. This is illustrated on Prog.1 and Prog.2, which respectively shows a CAL actor (computing a 1D thresholded gradient in this case) in its original form and after modifications. The additional modification regarding Flag in and Flag out, shown in Prog.2, is generic for all actors of the network. Regarding additional input port for taking into account parameters, this depends of each actor. If an actor needs an external parameters, so, additional input port is added, like in Prog.2 for the Threshold input port.

IV. EXPERIMENTAL RESULTS

IV-A. The "SeeMOS" smart camera platform

To validate the proposed approach, an heterogeneous smart camera, named SeeMOS, is used. The SeeMOS hardware architecture is presented in Fig. 6. This architecture is designed around FPGA, more precisely an Altera Stratix EP1S60. The FPGA device plays the central role in the system, being responsible to interconnect all other hardware devices. Surrounding it, 10Mb (5x2) of SRAM and 64Mb of SDRAM are available for image and other data storage. More details about SeeMOS platform can be found at [9], [10]. The SeeCORE described in Section 2 has been implemented on the SeeMOS platform.
IV-B. Results

Based on a previous work [11], the Lucas and Kanade optical flow computation algorithm [12] has been re-designed following the proposed approach. A comparison, in term of lines of code (loc), between a HDL hand written version and the presented methodology is reported in Tab. I. Tab. II shows synthesis report for an image resolution of 800x600. The soft-core occupation, without PEs, is less than only 4% of the FPGA (for a Stratix I) which allows large applications implementations. In term of frequency, the maximum achievable for the PEs designed with our methodology is 135 MHz.

<table>
<thead>
<tr>
<th></th>
<th>HDL hand written</th>
<th>Our Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control and schedule</td>
<td>950 loc</td>
<td>354 loc (assembly)</td>
</tr>
<tr>
<td>PEs</td>
<td>593 loc</td>
<td>374 loc (CAL)</td>
</tr>
</tbody>
</table>

Tab. I. Comparison between HDL hand written and proposed methodology version.

V. CONCLUSION AND FUTURE WORKS

In this paper a high-level design methodology for FPGA-based smart cameras is presented. Reported results shown in Tab. I and Tab. II are promising. Thus, the proposed methodology seems to be a good trade-off between efficiency and simplicity of programming. A future work is to develop a simulator of the SeeMOS platform on which the designer will describe the application with a CAL actors
library. Then, based on this work and after simulation validations, a compiler will provide the assembly code and the processing elements in HDL to be used on the SeeMOS platform.

VI. REFERENCES


<table>
<thead>
<tr>
<th>Part</th>
<th>Number of LEs</th>
<th>Memory bits</th>
<th>DSP blocks</th>
<th>Maximum Clock Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control and Schedule</td>
<td>364 (&lt; 1%)</td>
<td>73,728 bits(1%)</td>
<td>0</td>
<td>223 MHz</td>
</tr>
<tr>
<td>Peripheral drivers</td>
<td>696 (1%)</td>
<td>0</td>
<td>0</td>
<td>40 MHz (Communication driver)</td>
</tr>
<tr>
<td>Crossbar device</td>
<td>1191 (2%)</td>
<td>0</td>
<td>0</td>
<td>39 MHz (transition frequency)</td>
</tr>
<tr>
<td>Processing Elements</td>
<td>8099 (13%)</td>
<td>140,112 bits(3%)</td>
<td>54 (38%)</td>
<td>135 MHz</td>
</tr>
</tbody>
</table>

Table II. Synthesis report.

Program 1 Example of a CAL actor

```cal
actor gradient_1D(threshold)
    pxl_m, pxl_p => gdt:
    res := 0;
    action
    pxl_m:[prev], pxl_p:[past] => gdt:[res]
    do
        if past - prev > threshold or prev - past > threshold
        then res := past - prev;
        else res := 0;
    end
end
```

Program 2 Example of a CAL actor ready to be used in our programming environment

```cal
actor gradient_1D()
    int(size = 8)pxl_m, int(size = 8)pxl_p,
    int(size = 1)flag_in, int(size = 8)threshold
    int(size = 9)gdt,
    int(size = 1)flag_out:
    res := 0;
    action
    pxl_m:[prev], pxl_p:[past], flag_in:[busy], threshold:[thresh]
    => gdt:[res], flag_out:[busy]
    do
        if busy then
            if past - prev > threshold or prev - past > threshold
            then res := past - prev;
            else res := 0;
        end
    end
end
```