P2IP: A novel low-latency Programmable Pipeline Image Processor

Paulo Possa, Naim Harb, Eva Dokladalova, Carlos Valderrama

P^{2}IP: A novel low-latency Programmable Pipeline Image Processor ☆

Paulo Possa[a,*], Naim Harb[a], Eva Dokládalová[b], Carlos Valderrama[a]

[a] *Department of Electronics and Microelectronics, University of Mons Boulevard Dolez 31, 7000 Mons, Belgium*
[b] *Computer Science Laboratory Gaspard-Monge, ESIEE Paris, University Paris-Est 93162 Noisy-le-Grand Cedex, France*

Abstract

This paper presents a novel systolic Coarse-Grained Reconfigurable Architecture for real-time image and video processing called P^{2}IP. The P^{2}IP is a scalable architecture that combines the low-latency characteristic of systolic array architectures with a runtime reconfigurable datapath. Reconfigurability of the P^{2}IP enables it to perform a wide range of image pre-processing tasks directly on a pixel stream. The versatility of the P^{2}IP is demonstrated through three image processing algorithms mapped onto the architecture, implemented in an FPGA-based platform. The obtained results show that the P^{2}IP can achieve up to 129 fps in Full HD 1080p and 32 fps in 4K 2160p what makes it suitable for modern high-definition applications.

Keywords: Reconfigurable hardware, Image processing, Real-time system, Computer vision

1. Introduction

Digital image processing is a well-known class of computationally intensive tasks that conventional computing architectures cannot efficiently

☆This work is supported by the French Community of Belgium under the Research Action ARC-OLIMP (Optimization for Live Interactive Multimedia Processing 2008-2013)

*Corresponding author

Email addresses: paulo.possa@umons.ac.be (Paulo Possa), naim.harb@umons.ac.be (Naim Harb), e.dokladalova@esiee.fr (Eva Dokládalová), carlos.valderrama@umons.ac.be (Carlos Valderrama)

perform in terms of power consumption and when real-time performance is required [1, 2, 3, 4]. Real-time requirements are especially important in latency-critical applications, such as live video broadcasting, video surveillance, Unmanned Vehicle (UV) navigation, interactive multimedia applications, and video assisted medical devices. Latency is defined here as the amount of time between when a signal is impressed on the input of a circuit and when it is received or detected at its output [5].

We can divide image processing tasks into two groups: high-level and low-level tasks [6]. High-level image processing tasks are algorithms that create symbolic representations of the image contents, e.g., object recognition and tracking. Low-level image processing tasks are algorithms that can modify specific aspects in the image (e.g., color correction, filtering, and blurring), or detect features (e.g., edge and corner detectors). The data input of the last group is frequently a stream of a large collection of small and independent data elements [7]. Due to this inherent data parallelism, low-level tasks are very suitable for highly parallel processing architectures. Numerous implementations have been proposed in the past for this last group. They mainly target general flexibility improvement but the problem of minimal latency has not been directly studied.

In order to explore the data parallelism of low-level image processing tasks, we are proposing a new reconfigurable architecture, the Programmable Pipeline Image Processor (P^2IP). The P^2IP is a Coarse-Grained Reconfigurable Architecture (CGRA) based on a linear systolic array model targeting low-latency real-time applications and supporting a wide range of image resolutions and operators. Figure 1 shows a simplified functional diagram of the P^2IP architecture. Each Processing Element (PE) of the P^2IP contains an optimized set of essential image processing operators that can be parametrized at runtime. A Recongurable Interconnection module associated to each PE enables dynamic datapath re-routing. Configuration registers define the behavior of the operators distributed in the P^2IP including the Reconfigurable Interconnection. These registers can be accessed through a dedicated control path in order to change the application context without resynthesizing the system. Algorithms can be created or invoked by using a MATLAB embedded library that supports algorithm allocation and task mapping into the reconfigurable architecture through the configuration port of the P^2IP controller.

The parametrizable PEs and the reconfigurable datapath provide *post-synthesis* flexibility to the P^2IP architecture. Synthesis is defined here as

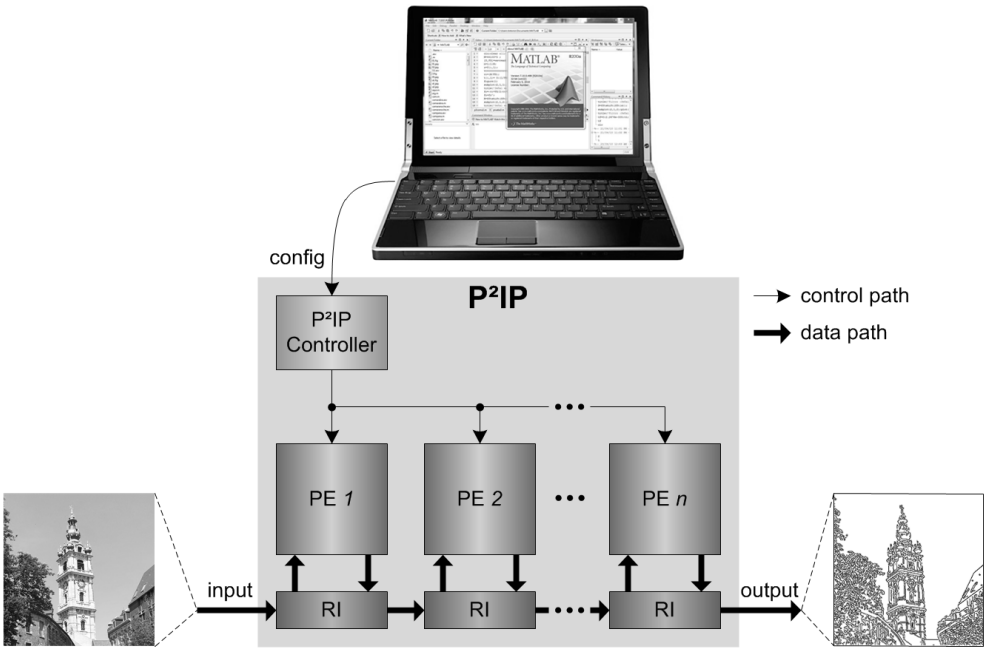

Figure 1: Simplified functional diagram of P^{2}IP architecture showing the datapath formed by Processing Elements (PEs) associated to a Reconfigurable Interconnection (RI).

the process where a digital circuit behavior, usually described in a Hardware Description Language (HDL) such as Verilog HDL or VHDL, is converted by a *synthesizer* into an association of logic gates that can be implemented in hardware. Thus, post-synthesis flexibility means that the P^{2}IP can still be configured to change its functionality after its hardware implementation. Beyond that, the modular organization of P^{2}IP facilitates pre-synthesis customizations in order to meet application requirements. That includes customizations such as the number of PEs, maximum frame resolution, and set of operators.

The rest of this paper is organized as follows: Section 2 presents related works. Section 3 describes the P^{2}IP architecture in detail. Examples of algorithms mapping are presented in Section 4. Afterwards, the algorithm performances and the architecture implementation results are analyzed in Section 5. Finally, the conclusions of this work are presented in Section 6.

2. Related work

It has been almost half a century since one of the first digital computers specially designed for image processing started operating [8]. Since then, a large number of specialized architectures have been designed with the same basic purpose: to process images automatically and efficiently. Historically, the performance requirements of image processing applications have only been met with special-purpose (custom), fixed-function hardware [3, 9], i.e. Application-Specific Integrated Circuits (ASICs) [10]. However, due to the lack of flexibility in custom solutions, alternatives have been proposed by the community in order to provide more flexible architectures. Among these alternatives, three computing models can be highlighted: Domain-specific processors, e.g., Digital Signal Processors (DSPs) [11, 12], Single-Instruction Multiple-Data (SIMD) architectures [1, 13, 14], and Reconfigurable Computing (RC) systems [10, 15, 16].

Among these alternatives, the most promising in terms of computing performance and energy efficiency is the RC approach [15, 17, 18]. RC systems can be divided into two groups, Fine-Grained Reconfigurable Architectures (FGRAs) and CGRAs. FGRAs, e.g., Field Programmable Gate Arrays (FPGAs), are extremely flexible architectures containing PEs in which it is possible to map any 1-bit logic function at the cost of long design and compilation times. CGRAs overcome this cost by using more elaborated PEs that can perform word-level operations for a wide range of applications including image and video processing. Some important examples of these architectures are PipeRench [19], RAW [20], and MORPHEUS [17]. PipeRench is a linear array accelerator for pipelined applications composed of 256 ALU-based PEs that can be reconfigured dynamically. RAW is a multiprocessor architecture containing 16 MIPS-style microprocessors arranged in a 4×4 array interconnected by a both static and dynamic network. MORPHEUS is a more recent architecture including three heterogeneous processing engines, each one with a different reconfigurable granularity, interconnected by a Network-on-Chip (NoC) and controlled by a RISC processor. However, these solutions generate supplementary latency, penalizing the execution time with respect to fixed-function hardware.

In our approach, we combine the flexibility of RC systems to the processing performance and latency of fixed-function hardware solutions to bridge their performance/flexibility gap. The P^2IP flexibility was obtained by exploring classic low-level image processing algorithms, such as Canny Edge

Detection and Harris Corner Detection, resulting in the design of fundamental building operators that could be generalized and combined in PEs to support a wide range of algorithms. The cost of the P^{2}IP approach is to be restricted to a single application domain which in this work corresponds to image and video pre-processing. However, this characteristic reduces algorithm mapping complexity since many image processing operators are already implemented on the architecture, e.g., linear filters and non-maximum suppressors.

Other architectures that offer competitive performances targeting low-power image processing applications are CSX700 [21], Diet SODA [14], and CRISP [22]. CSX700 presents a low-power SIMD architecture for image processing, containing 192 PEs divided into two processing cores. Diet SODA is another low-power architecture targeting Digital Still Cameras (DSCs), containing a 128-lane SIMD unit that works at low frequencies. CRISP processor is a CGRA, but optimized for image and video processing in DSCs. It presents a series of specific image processing operators associated to a configurable interconnection that routes the data stream through these operators, which makes it possible to change the processing task.

It is also interesting to cite here another popular approach for image processing based on General-Purpose Graphics Processing Units (GPGPUs). Though these solutions do not target low power consumption application, they have presented very competitive performances [1, 13]. Also, the work in [23] explores a parallel hybrid approach using optimized heterogeneous multi-CPU/multi-GPU architectures to address image processing tasks.

3. P^{2}IP architecture

The P^{2}IP architecture can be classified as a CGRA based on a linear systolic array model. Images or frames are entered as a stream of pixels in sequential line-scanned format progressing through the pipeline at a constant rate. The P^{2}IP datapath works at the pixel clock frequency and can deliver one processed pixel per clock cycle after the initial latency to fill the pipeline. It was designed to work between a frame source and a frame sink directly on the pixel stream, as shown in Fig. 2.

In order to permit the P^{2}IP integration into an image processing chain, the AXI4-Stream [24] was adopted as the external interconnection protocol. The AXI4-Stream protocol is controlled by the P^{2}IP Controller which is also in charge of reading configuration words on the configuration input

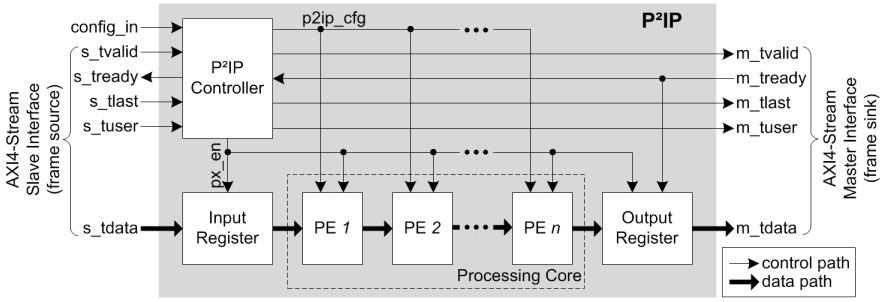

Figure 2: Functional diagram of the P^{2}IP architecture.

port (*config_in*) and their transfer to the PEs. The P^{2}IP Controller is the input node of the P^{2}IP configuration mechanism formed by the *Configuration Tree*. A detailed description of the P^{2}IP configuration mechanism is given in Section 3.2.

The input register block along with the output register block (Fig. 2) constitute the boundaries of the P^{2}IP architecture. This practice prevents any timing critical changes due to nets crossing the module boundaries. Also, it helps synthesis tools to optimize a module without the interference of other system components [25, 26]. The input register block separates three color channels (red, green, and blue) from a true-color input data stream (24 bits per pixel) before transferring it to the first PE in the pipeline. It also generates a fourth color channel (8 bits per pixel) carrying a grayscale version of the input pixel stream. A grayscale image represents an effectively continuous range of tones, from black to white, through intermediate shades of gray [27].

The processing core of the P^{2}IP is formed by a sequence of identical PEs. Although the PEs have the same internal structure, each one can perform a different function according to its configuration. The number of PEs can be defined before synthesis by a simple instantiation procedure without changing the P^{2}IP basic structure or the programming mechanism. Following, the P^{2}IP PE is described in detail.

3.1. The P^{2}IP Processing Element

The PEs are the main components of the P^{2}IP. All PEs are identical, formed by a Reconfigurable Interconnection and modules containing groups of image processing operators. The Reconfigurable Interconnection directs

the pixel stream through modules forming the P^{2}IP datapath. Fig. 3 shows the PE functional diagram.

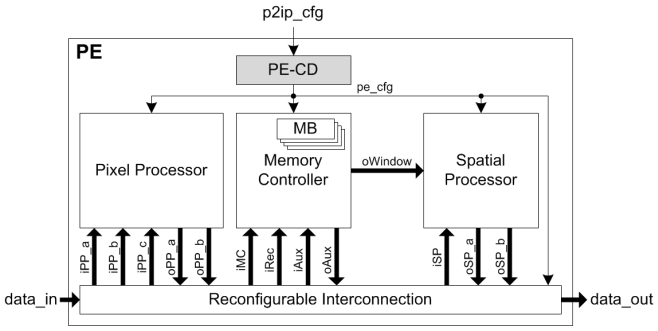

Figure 3: Functional diagram of the P^{2}IP Processing Element (PE).

Each PE has its own local data memory divided into 4 Memory Blocks (MBs) used for storing frame lines (one line per MB). The maximum horizontal size of the input frame is limited by the MB's size which can be defined before synthesis. The MBs are based on dual-port memory elements accessed sequentially. The *Memory Controller* performs all memory operations needed by the PE, e.g., neighborhood extraction and delay. The other modules, *Pixel Processor* and *Spatial Processor*, comprise clusters of image operators. The Pixel Processor module contains pixel-to-pixel operators which process each pixel individually, e.g., arithmetic and logic operations. The Spatial Processor module contains spatial operators which are neighborhood oriented functions. These functions require an input window formed by the targeted pixel and its neighborhood to provide a result. Those three modules are interconnected by the Reconfigurable Interconnection. Also in Fig. 3, we can see the PE-Configuration Decoder (PE-CD) which is the second node of the Configuration Tree. The PE-CD distributes the configuration received from the P^{2}IP Controller to the internal PE modules. More details about this block and the P^{2}IP configuration mechanism are given in Section 3.2. Following, we present a detailed description of the PE internal modules.

3.1.1. Memory Controller

As mentioned earlier, the Memory Controller contains memory-based operators, more specifically, a Neighborhood Extractor (NE), a mirror, and a delay. Fig. 4 shows the functional diagram of the Memory Controller structure.

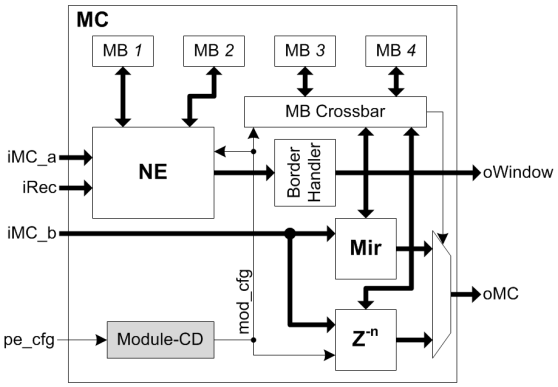

Figure 4: Functional diagram of the Memory Controller (MC).

In order to reduce memory resources, MB 3 and 4 (Fig. 4) can be allocated to any Memory Controller operator through the MB crossbar. It is important to highlight here that these MBs can be allocated to only one operator at a time. In this way, the NE can have up to four allocated MBs, allowing it to generate sliding windows of 5×9 pixels or 9×9 by associating two consecutive PEs. All data inputs and outputs of the Memory Controller module are connected to the Reconfigurable Interconnection, with the exception of the window output (*oWindow*) connected to the Spatial Processor module (Section 3.1.2) and the configuration input (*pe_cfg*) that is part of the Configuration Tree (Section 3.2). There follows a description of the memory-based operators.

- *Neighborhood Extractor (NE)*: An NE provides a sliding window that scans the whole image. This window contains a pixel neighborhood that is necessary for some computations, e.g., 2D Convolution. The operating principle of a 3×3 neighborhood extractor is presented in Fig. 5. The line buffer size of the NE operator can be configured after synthesis in order to adjust it to the frame size. In addition, the number of line buffers can be configured, from 2 to 4 by allocating more MBs. The number of elements in a pixel array is fixed with nine registers. Thus, the NE operator can support sliding windows of up to 5×9 pixels. The NE operator also supports recursive operations by introducing a second pixel stream directly into the line buffer located just after the central pixel array. This feature allows the PE to process a part of the window twice, what can reduce the number of operators

in certain computations. Also, in order to avoid random pixels in subsequent stages, a border handler is placed at the output of the NE block (Fig. 4). The border handler assigns determined values to pixels of the window that crosses the image's border. This situation occurs when the center of the window is located near to the image border and part of the window stays outside the image boundaries. In our system, we replicate the inner window pixel values to those pixels outside the image boundaries.

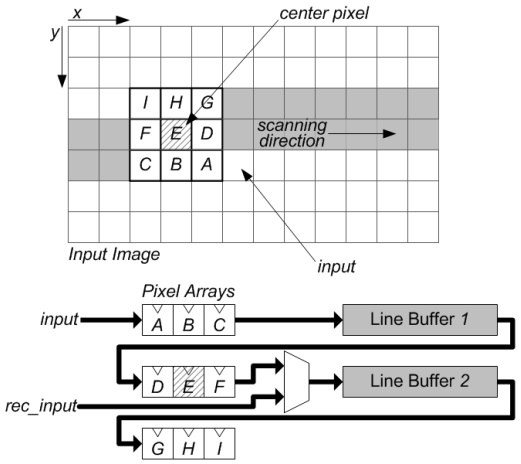

Figure 5: Operating principle of a 3×3 neighborhood extractor.

- *Mirror (Mir*: The function of the Mirror operator is to invert the direction in which the image is scanned. This is necessary when an algorithm must propagate an event in the image and the regular scanning direction does not allow it. The Mirror structure is based on two line buffers working in a Last-In-First-Out (LIFO) fashion.

- *Delay (Z^{-n})*: The Delay operator is composed of two line buffers with configurable size and is used to synchronize two pixel streams that are in different stages of the P^2IP. The maximum delay supported is 2 frame lines which is equivalent to the delay imposed by a 5×5 neighborhood operation.

3.1.2. Image operators

The Spatial Processor is a PE module containing three fundamental neighborhood-based operators for low-level image processing, more specif-

ically, a Two-Dimensional Convolver (2DC), a Non-Maximum Suppressor (NMS), and a connector. Fig. 6 shows the functional diagram of the Spatial Processor module.

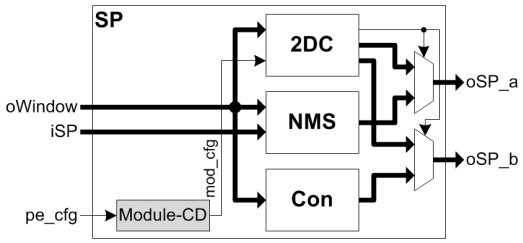

Figure 6: Functional diagram of the Spatial Processor (SP).

The output window from the Memory Controller module (Section 3.1.1) is the main input of the Spatial Processor module. The other inputs and outputs are connected directly to the Reconfigurable Interconnection, with the exception of the configuration input that is part of the Configuration Tree. The 2DC configuration selects which operator will be activated in the module. Next, we present a description of the Spatial Processor operators.

- *Two-Dimensional Convolver (2DC)*: The 2DC is a fundamental operator in many image processing algorithms. It can convolve an input image $f(x, y)$ with a predefined kernel or mask $h(i, j)$ such as Sobel, Gaussian, and Laplacian, as defined in (1).

$$g(x, y) = \sum_{i,j} f(x + i, y + j) \cdot h(i, j) \tag{1}$$

  where $g(x, y)$ is the resulting image, $f(x, y)$ is the input image, and $h(i, j)$ is the kernel. The equation above can be more compactly noted as in (2).

$$g(x, y) = f(x, y) \otimes h(i, j) \tag{2}$$

  A set of these kernel coefficients is stored in a local ROM block. They can be downloaded during the 2DC configuration step. An important feature of the 2DC is that it can compute up to two 3×3 kernels in parallel per module or a single 5×5 kernel. This approach can reduce idle resources in certain operations.

- *Non-Maximum Suppressor (NMS)*: The NMS operator analyses the input window and outputs the center pixel only if it is the local maximum. Otherwise, NMS outputs the value zero. It has a specific input that provides the gradient direction of the input window. This direction is estimated by the direction operator in the Pixel Processor module.

- *Connector (Con)*: The connector operator is used during edge detection to reduce the number of gaps along lines representing edges. It analyses the neighborhood of a pixel to verify if it can be connected to another pixel in its neighborhood.

The Pixel Processor is a PE module containing a set of optimized pixel-to-pixel operators. Fig. 7 depicts the Pixel Processor module showing all internal operators and their respective interconnection. Following, we give an overview of these operators.

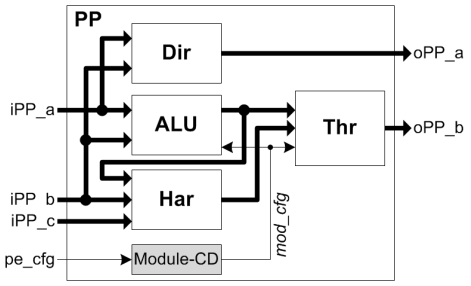

Figure 7: Functional diagram of the Pixel Processor (PP).

- *Direction (Dir)*: This operator compares if the input *iPP_a* value is greater than *iPP_b* and its output (*oPP_a*) is Boolean. It is used as an approximation of the arctangent function which provides the direction of a gradient.

- *Harris (Har)*: The Harris operator computes the Harris response based on [28]. The Harris response is an alternative to calculate the eigenvalues for feature detection applications.

- *Arithmetic and Logic Unit (ALU)*: The ALU operator is a more generic and configurable operator that can compute the following functions: multiplication, square power, binary shifting, addition, subtraction, logic AND (&), greater than, and less than.

- *Threshold (Thr)*: The Threshold operator performs a simple segmentation technique, classifying pixels into two categories according to rule (3)

$$g_{th}(x,y) = \begin{cases} 0, f(x,y) < T \\ 1, f(x,y) \geqslant T \end{cases} \tag{3}$$

where $g_{th}(x,y)$ is the resulting image of the threshold operator, $f(x,y)$ is the image function, and T is the threshold limit value. It can also be configured to operate in a hysteresis mode where two threshold limits (T_{low} and T_{high}) can be used according to rule (4).

$$g_{th}(x,y) = \begin{cases} 0, f(x,y) < T_{low} \\ f(x,y), T_{low} \leqslant f(x,y) < T_{high} \\ 1, f(x,y) \geqslant T_{high} \end{cases} \tag{4}$$

The two clusters of image processing operators in each PE of the P^{2}IP can perform a series of fundamental computations and combinations of them. To illustrate some of the operations supported by the P^{2}IP, Table 1 shows the relations between the P^{2}IP operators and low-level image processing operations.

Table 1: Relation between the P^{2}IP operators and low-level image processing operations.

Low-level Operations	P^{2}IP Operators						
	2DC	NMS	Con	Dir	ALU	Thr	Har
Thresholding						x	
Arithmetic Operations					x		
Smoothing	x						
Sharpening	x				x		
Noise Reduction	x						
Edge Detection	x	x	x	x	x	x	
Corner Detection	x	x			x	x	x
Gradient Direction	x			x			
First-order Derivative	x				x		
Laplacian	x						
Basic Segmentation						x	

3.1.3. Reconfigurable Interconnection

The Reconfigurable Interconnection is the PE module in charge of routing different datapaths in the P^{2}IP architecture. It is composed of two independent configurable crossbars with 8-bit wide paths. Each external input or output of the Reconfigurable Interconnection passes by a register. This approach guarantees that the maximum pixel clock supported by the P^{2}IP does not change with long paths between PEs. Fig. 8 shows the Reconfigurable Interconnection functional diagram.

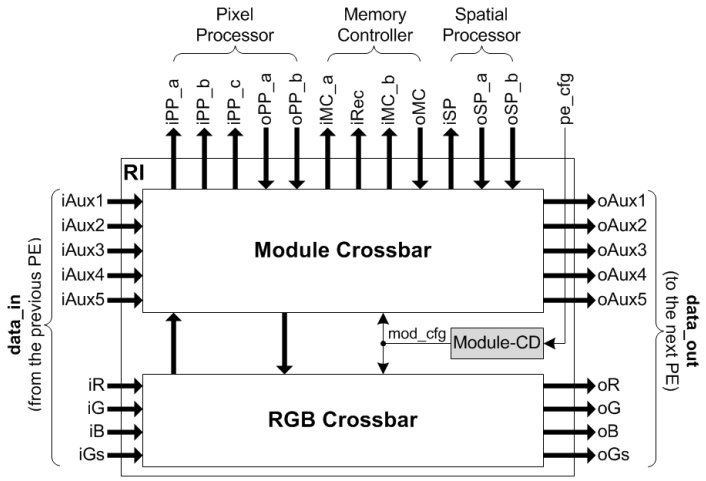

Figure 8: Functional diagram of the Reconfigurable Interconnection (RI).

The module crossbar interconnects all modules in a PE. Its auxiliary I/Os (*iAux* and *oAux*) can share partial results directly with other PEs. The RGB crossbar is the main I/O of a PE. As the P^{2}IP can only process one color component per PE, the RGB crossbar has a single I/O channel connected to the module crossbar. It outputs a selected color component to be processed by the PE and receives the result which is transferred to the next PE. Fig. 9 shows three basic configurations that can be associated to obtain different datapaths.

The pipeline configuration (Fig. 9a) is the natural datapath supported by the P^{2}IP architecture. In this model, the input data passes through a cascaded series of Operators (Op). In the second configuration (Fig. 9b), the input data is processed simultaneously in two different PEs and their outputs are processed in a third PE. The third configuration, presented in Fig. 9c, has three concurrent inputs that are processed simultaneously in three different

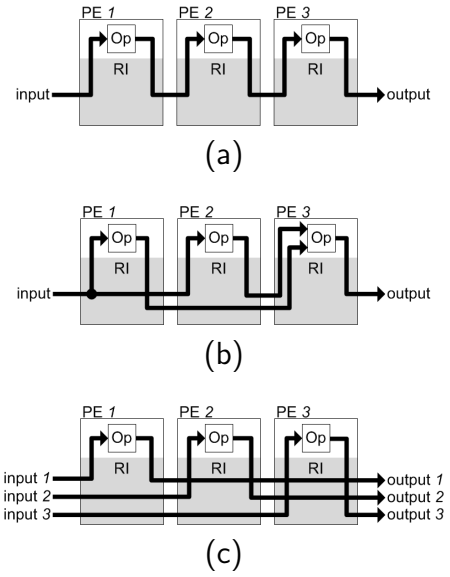

Figure 9: Basic datapath configurations mapped through the Reconfigurable Interconnection (RI). (a) Regular pipeline of Operators (Op). (b) Fork then fusion. (c) Independent parallel pipelines.

PEs. This last configuration is mostly used for color processing.

3.2. Configuration mechanism

The Configuration Tree is a scalable configuration structure formed by a series of Configuration Decoders (CDs) hierarchically organized. The objective of the Configuration Tree is to provide access to all configuration registers located in the P^{2}IP operators. Fig. 10 depicts the Configuration Tree architecture showing all decoder levels, from the configuration input (*config_in*) to the configuration register.

The configuration input of the Configuration Tree is an 8-bit wide bus that can be connected to any kind of external interface, e.g., UART, Ethernet, or flash memory. Differently from the datapath that works at the same clock frequency as the input pixel stream, the Configuration Tree works at a fixed clock frequency of 100 *MHz*. An input configuration word takes four clock cycles to arrive at a specific operator. Each configuration process can send a maximum of eight configuration bytes to a determined operator. Considering the extra two bytes corresponding to the configuration header (operator location and number of extra bytes), an operator takes a maximum of 0.34 μs to be configured.

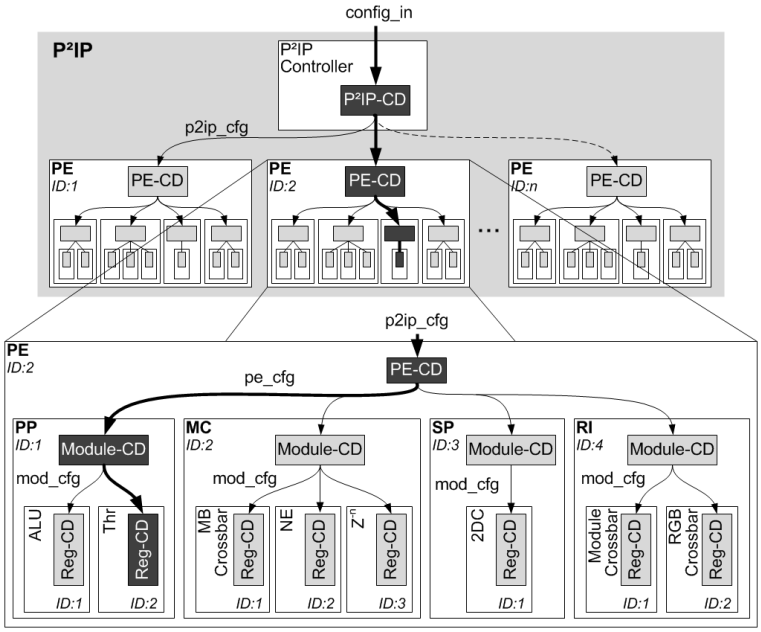

Figure 10: Configuration Tree architecture with a highlighted path showing the active modules during an operator configuration.

The P²IP-CD is the input node of the Configuration Tree. It extracts global parameters from the configuration input, such as the frame size, and transfers the rest to the next level of decoders formed by PE-CDs. Each PE and its internal components have an Identification (ID) number. These IDs are used to identify operators in the P²IP architecture. To access an operator, firstly it is necessary to send a header through the configuration bus containing all IDs in the path towards the operator. During the configuration process, the PE-CDs distribute the configuration words to the next level of decoders inside the processing modules, called Module Configuration Decoders (Module-CD). The Module-CDs send the configuration words to the final level of decoders located inside the processing operators, the Register Configuration Decoders (Reg-CDs). The configuration registers are located in the Reg-CDs and they vary in size depending on how many parameters must be set to configure the operator. Fig. 11 shows the register fields of the configuration header and an example of a configuration register to configure a Threshold operator.

In Fig. 11, the Threshold configuration register is composed by three

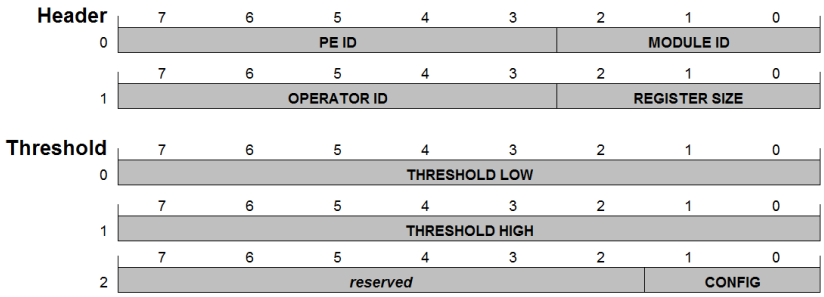

Figure 11: Register fields of the configuration header and the Threshold configuration register.

bytes where the first byte configures the lower threshold limit, the second byte configures the higher threshold limit, and bits 0 and 1 of the third byte indicate the operating mode of the Threshold operator. The Threshold operator can be configured to work in three different modes:

1. Bypass;
2. Normal mode as in eq. (3) where only the lower threshold limit is taken in consideration;
3. Hysteresis mode as in eq. (4) where T_{low} is the lower threshold limit and T_{high} higher threshold limit.

The configuration transfer at the configuration input port (*config_in*) is accepted when the corresponding *VALID* signal is high. Fig. 12 presents an example of configuration transfer.

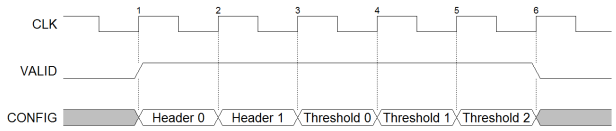

Figure 12: Example of configuration transfer targeting a Threshold operator.

The methodology to configure the P^{2}IP by manually creating configuration transfer sequences is timing consuming and error prone. In order to facilitate the P^{2}IP configuration process, a function library embedded on MATLAB-based language has been created. This library accepts mnemonics to cover all possible configurations of the P^{2}IP operators, giving to the user full control over the architecture. In addition to creating configuration transfer sequences, MATLAB allows the user to send them directly to the

P^{2}IP by creating *Interface Objects* (e.g. UART, UDP, I^{2}C, and Bluetooth). Following, we give an example of configuration using the function library:

```
>> P2IP(ser, 'PE',2, 'threshold', 'Tcfg',3, ...
   'Ta',50, 'Tb',100)
```

where *P2IP(...)* is the main MATLAB function that calls the internal functions; *ser* is a MATLAB opened communication port that will be used to transfer the configuration words generated by the assembler; *'PE'* is the mnemonic for PE ID that receives the value 2; and *'threshold'* is the mnemonic for the threshold operator that receives three parameters: $Tcfg = 3$ corresponding to the hysteresis operating mode; $Ta = 50$ corresponding to T_{low} value; and $Tb = 100$ corresponding to T_{high} value.

4. Algorithm partitioning and allocation

The objective of algorithm partitioning is to divide the algorithm into groups of operations that can fit in a single PE and analyze the appropriate configuration of the interconnection. The basic rule for an efficient algorithm mapping on the P^{2}IP is to allocate at least one neighborhood operator per PE and distribute the remaining operators according to the available resources and their data dependencies.

In order to validate the proposed architecture, three low-level image processing algorithms have been mapped on the P^{2}IP, more specifically, the Edge Sharpening, Canny Edge Detection, and Harris Corner Detection algorithms. These algorithms are very popular in the field and used in a wide range of applications, e.g. image enhancement, image segmentation, object recognition, and object tracking. In addition, they require a variety of operators that offer an interesting use case for evaluating our approach. Following, we give an overview of these application algorithms and then a description of the partitioning of each algorithm.

4.1. Edge Sharpening

Edge Sharpening is a technique for enhancing edges in images with a low level of edge definition, e.g. X-ray images. It commonly uses the unsharp sharpening algorithm where an unsharp filter extracts the image gradient edges and then adds them back onto the original image [29]. Mathematically,

the unsharp filter produces an edge image $e(x, y)$ from an input image $f(x, y)$, as defined in (5).

$$e(x, y) = f(x, y) - f_{smooth}(x, y) \tag{5}$$

where $f_{smooth}(x, y)$ is a smoothed version of $f(x, y)$ that must be previously computed.

To obtain the sharpened edges, the result of (5) is added back to the original image, as in (6).

$$s(x, y) = f(x, y) + k \cdot e(x, y) \tag{6}$$

where $s(x, y)$ is the resulting image with sharpened edges, and k is a scaling constant.

A different commonly used technique is to apply a negative Laplacian kernel (7) as the unsharp filter, simplifying the algorithm in one operation. Using this technique, the unsharp filter $e(x, y)$ can be defined as in (8).

$$h_l(i, j) = \frac{1}{16} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{7}$$

$$e(x, y) = f(x, y) \otimes h_l(i, j) \tag{8}$$

Fig. 13 presents the Edge Sharpening algorithm graph for a single color channel applying a Laplacian kernel (h_l) as the unsharp filter and its implementation on the P^{2}IP architecture in a true-color (24-bit RGB) format.

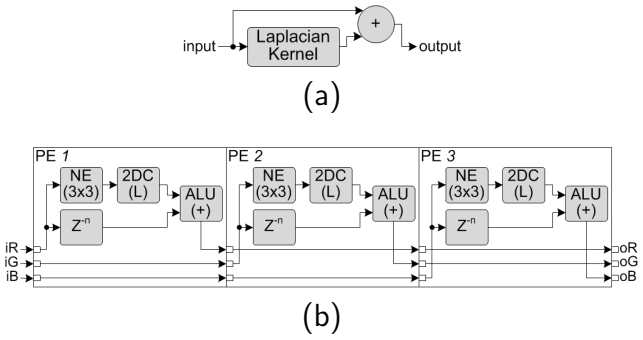

Figure 13: Edge Sharpening algorithm. (a) Algorithm graph. (b) P^{2}IP implementation.

The Edge Sharpening implementation (Fig. 13b) requires three PEs for a true-color format and performs 60 operations per pixel along the datapath.

The datapath configuration is similar to the one presented in Fig. 9c where each PE has an identical set of operators working in parallel. For each color channel, the input is computed by a 3×3 Laplacian (L) kernel and the ALU operator adds its result to a delayed version of the input provided by the delay (Z^{-n}) operator.

4.2. Canny Edge Detection

Edge detection is a fundamental process in computer vision for image segmentation and object recognition. Edges are prominent events due to local changes in intensity or color in images [30]. Basically, if the brightness of a pixel is significantly different from the pixels in its neighborhood, it may contain an edge. It is possible to detect such changes in an image $f(x, y)$ by applying special kernels, such as the Sobel kernel (9). These kernels are presented in pairs where one is used to detect horizontal variations (h_h) on the image and the other vertical variations (h_v). As a result of these operations, we obtain a horizontal gradient $g_h(x, y)$ and a vertical gradient $g_v(x, y)$ as defined in (10) and (11), respectively.

$$h_h(i,j) = \frac{1}{8}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } h_v(i,j) = \frac{1}{8}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{9}$$

$$g_h(x,y) = f(x,y) \otimes h_h(i,j) \tag{10}$$

$$g_v(x,y) = f(x,y) \otimes h_v(i,j) \tag{11}$$

With these gradients, it is possible to obtain a gradient magnitude or modulus that represents the local edge strength $|G(x, y)|$ (12) and the local edge orientation angle Φ (13) using an arctangent operation.

$$|G(x,y)| = \sqrt{g_h(x,y)^2 + g_v(x,y)^2} \tag{12}$$

$$\Phi(x,y) = tan^{-1}\left(\frac{g_v(x,y)}{g_h(x,y)}\right) \tag{13}$$

Canny Edge Detection is one of the most popular algorithms for edge detection due to its minimum number of false edge points, good localization of edges, and single mark on each edge [30]. The Canny algorithm is composed of three steps: smoothing, edge enhancement, and localization. For the

smoothing step, the Canny algorithm uses a Gaussian low pass filter, based on a Gaussian kernel (14), to suppress the noise of the input image.

$$h_g = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \tag{14}$$

Next, during the edge enhancement step, a gradient vector at each pixel of the smoothed image is calculated based on the Sobel kernel, as defined in (9). The localization step is divided into two stages: NMS and Hysteresis Thresholding (HThr). The objective of the NMS is to eliminate non-ridge pixels giving a one pixel wide aspect at the edges. A ridge pixel is defined as a pixel with a gradient magnitude greater than that of the adjacent pixels in the direction of the gradient. In the HThr stage, two thresholds are used: T_{low} and T_{high}. All pixels with a magnitude higher than T_{high} are considered as true edges. Pixels with magnitudes between T_{high} and T_{low} are considered as edge candidates. Pixels that do not satisfy these two criteria are suppressed. Edge candidates become true edges if they are connected to true edges directly or through other candidates. Fig. 14a presents the algorithm graph of the Canny Edge Detection, while Fig. 14b shows how the algorithm is partitioned and implemented on the P^{2}IP architecture.

The Canny Edge Detection implementation (Fig. 14b) works with a flow of 8-bit grayscale pixels. It starts by computing a 5×5 Gaussian (G) kernel. In *PE2* two 3×3 kernels are computed in parallel, the horizontal Sobel (S_H) and the vertical Sobel (S_V). Also in this PE, the gradient direction is estimated by the direction (Dir) operator and the ALU computes a gradient magnitude approximation based on [29]. The direction is transferred by an auxiliary channel (*oAux1*) to the next PE where it is used to process the NMS. Finally, the threshold (Thr) operator of the *PE3* along with *PE4* and *PE5* perform an approximation of the HThr operation where a sequence of connectors (Con) and mirrors (Mir) can fill most of the gaps in edge lines. The datapath structure in this implementation is mostly a regular pipeline as presented in Fig. 9a. The mapped version occupies five PEs and performs 123 operations per pixel along the datapath.

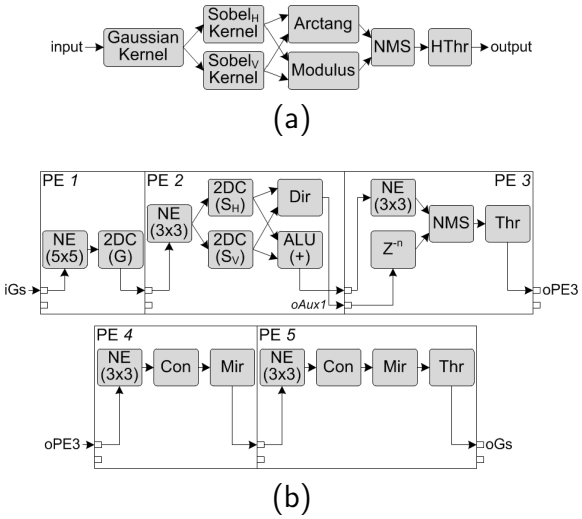

Figure 14: Canny Edge Detection algorithm. (a) Algorithm graph. (b) P^{2}IP implementation.

4.3. Harris Corner Detection

Corner detection is another fundamental process in computer vision. It is mainly used for motion detection, object tracking, panorama stitching, and 3D modeling. A corner is defined as an area that exhibits a strong gradient value in multiple directions at the same time [30]. The Harris operator uses this premise to find corners on an image. The first step is to obtain the first partial derivative of the image function $f(x, y)$ in both directions, horizontal and vertical, based on approximations (15) and (16).

$$g_H(x, y) = \frac{\delta f}{\delta x}(x, y) \approx f(x, y) \otimes \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{15}$$

$$g_V(x, y) = \frac{\delta f}{\delta y}(x, y) \approx f(x, y) \otimes \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \tag{16}$$

With the values of $g_H(x, y)$ and $g_V(x, y)$, it is possible to calculate the elements of the matrix M, described in (17), using (18), (19), and (20).

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix} \tag{17}$$

$$A = g_H(x, y)^2 \otimes w \tag{18}$$

$$B = g_V(x,y)^2 \otimes w \tag{19}$$

$$C = (g_H(x,y) \cdot g_V(x,y)) \otimes w \tag{20}$$

where w is a smoothing circular operator, e.g., a Gaussian kernel as defined in (14). The final step is to obtain the Harris operator response as in (21).

$$R = \mathrm{Det}[M] - k \cdot \mathrm{Tr}^2[M] \tag{21}$$

where R is positive in corner regions, negative in edge regions, and small in flat regions, k is a constant coefficient that, in practice, is a fixed value in the range of 0.04 to 0.06, and Det and Tr are the *determinant* and *trace* matrix operations, respectively. An optional final step is to select the best results in a determined region in order to reduce false corners. It can be done by a 10×10 NMS operator. Fig. 15a presents the algorithm graph of the Harris Corner Detection, while Fig. 15b shows how the algorithm is partitioned and implemented on the P^{2}IP architecture.

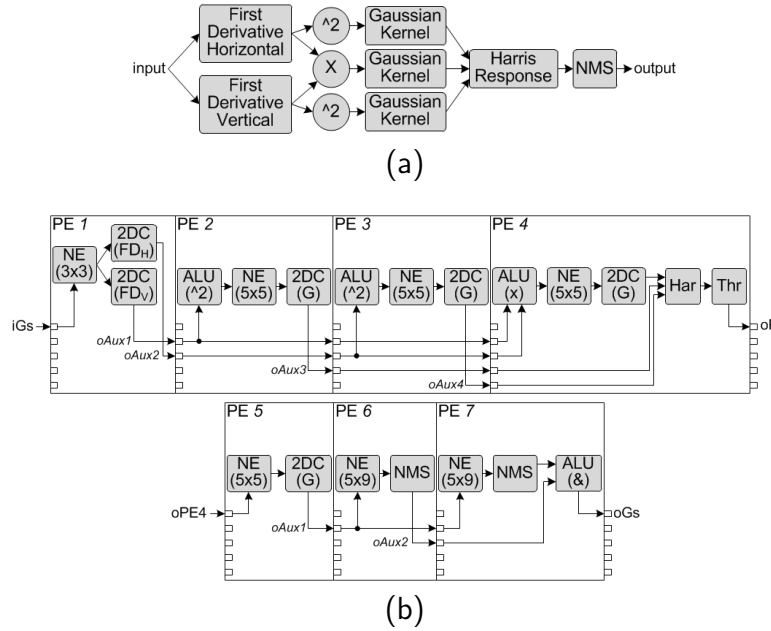

Figure 15: Harris Corner Detection algorithm. (a) Algorithm graph. (b) P^{2}IP implementation.

The Harris Corner Detection implementation on the P^{2}IP architecture (Fig. 15b) works with a flow of 8-bit grayscale pixels. It starts by computing

two 3×3 kernels in parallel, the horizontal first derivative (FD_H) and the vertical first derivative (FD_V). The output of the *PE1* is transferred to 3 different PEs (2, 3, and 4). In *PE4*, the Harris (H) operator uses the result of the previous 5×5 Gaussian kernel along with the results from *PE2* and *PE3* to compute the Harris response. The datapath structure along *PE2*, *PE3*, and *PE4* is an example of the model presented in Fig. 9b. In *PE5*, an extra 5×5 Gaussian kernel is computed in order to help the NMS operator during the selection of the best results. Finally, *PE6* and *PE7* create two 5×9 windows where one represents the top and the other represents the bottom part of a 9×9 window. The P^{2}IP version of the algorithm requires 7 PEs and performs 340 operations per pixel along the datapath.

5. Results

As mentioned earlier, latency is a critical characteristic of image processing systems in applications that must react as fast as possible to events captured by the image sensor. In the P^{2}IP, each PE can have a different latency according to its configuration. In order to simplify the latency analysis, we will assume a worst-case scenario, when a pixel stream takes the longest path along a PE. The longest path in a PE is shown in Fig. 16 where the corresponding latency is expressed in pixels since the P^{2}IP works at the input stream clock frequency, processing one pixel per clock cycle.

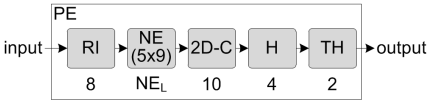

Figure 16: Longest path in a PE and the related latency. The latency is expressed in pixels.

The NE latency (NE_L) for a neighborhood window with $m \times n$ pixels in an image with dimensions $M \times N$ pixels, can be expressed as defined in (22).

$$NE_L = N\left(\frac{m-1}{2}\right) + \frac{n-1}{2} + b \tag{22}$$

where N is the number of pixels in one line of the input image and, for the neighborhood window, m is the number of lines, n is the number of pixels per line, and b is the Border Handler latency. As the largest window provided by

a single NE is 5×9 and the border handler latency is 3 clock cycles, we can obtain the PE total latency for the worst-case scenario (PE_L) as in (23).

$$PE_L = 2N + 32 \tag{23}$$

Finally, the total latency of the system can be expressed as in (24), taking into consideration that PEs operating in parallel are not included as active PEs.

$$P^2IP_L = p_a \cdot PE_L \tag{24}$$

where p_a is the number of active PEs in the P^{2}IP architecture.

In order to evaluate the P^{2}IP performance, we have implemented the architecture containing 10 PEs and supporting line buffers of up to 4095 pixels in an FPGA-based platform Altera Stratix IV EP4SGX230. The maximum clock frequency (F_{max}) in the datapath reported after synthesis was 268 *MHz*. As the P^{2}IP delivers one processed pixel per clock cycle, the datapath F_{max} corresponds to the architecture throughput that in this case is 268 Mpixel/s. Table 2 presents the P^{2}IP benchmarking obtained from HDL simulation targeting the low-level image processing algorithms described in the last section. Due to the constant output data rate, the throughput results do not change from one application to another. The performance varies according to the number of operations per pixel along the datapath, but stays constant for any resolution considering maximum throughput. Fig. 17 shows examples of images processed by the P^{2}IP using the algorithms mentioned.

Table 2: Throughput (T), Performance (P), and Latency (L) benchmarking results for Edge Sharpening (ES), Canny Edge Detection (CED), and Harris Corner Detection (HCD) algorithms w.r.t. P^{2}IP (10 PEs @ 268 *MHz*).

Application	Full HD[a]		4K[b]		Any
	T (fps)	L (μs)	T (fps)	L (μs)	P ($GOPS$)
ES	129	21.9	32	43.3	16.1
CED		43.5		86.6	33.0
HCD		65.1		129.5	91.1

[a] 1080×1920 pixels.
[b] 2160×3840 pixels.

Table 3 shows the FPGA resources required by the P^{2}IP architecture with 10 PEs. Considering that the P^{2}IP reconfigurability is mainly supported

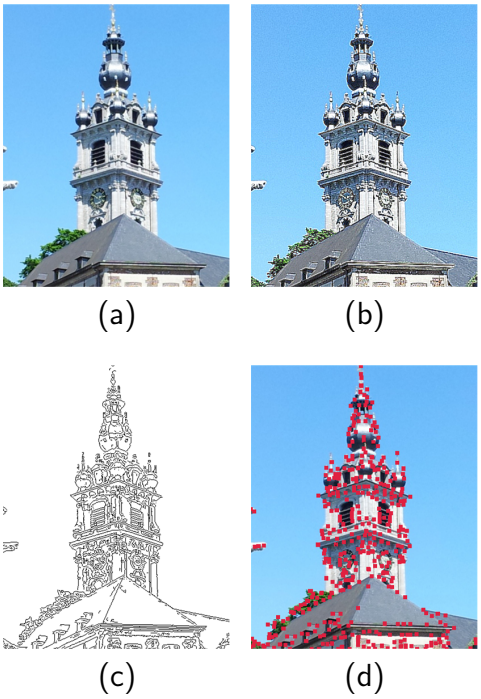

Figure 17: Example of images processed on the P^{2}IP architecture. (a) Original image - The belfry of Mons. (b) Edge Sharpening. (c) Canny Edge Detection - inverted output for better visualization. (d) Harris Corner Detection.

by the Configuration Tree and the Reconfigurable Interconnection, we can establish the amount of resource overhead generated only by this feature, corresponding to 15 % of the total resources. This overhead is more important when we take into consideration the resources left idle in comparison with a dedicate not-flexible implementation of the same algorithm. An example of this is when we compare a bare-metal implementation of the Harris Corner Detection with its analogous P^{2}IP implementation. The overhead in this case is an average of 74 % in each PE. If resource utilization is a design concern, the overhead can be drastically reduced by mapping as fixed PEs' parameters, all operators that are not required to be modified after-synthesis. During synthesis, the synthesizer in capable of optimizing these PEs by removing all resources that are not associated to the mapped operators, i.e., idle resources.

A comparison between the P^{2}IP performance and the state-of-the-art architectures is presented in Table 4. We can see that P^{2}IP has a better

Table 3: FPGA resources w.r.t. P^{2}IP (10 PEs).

Component	ALM[a]	Dedicated registers	Memory ($kbits$)	DSPs[b]
Controller	218	133	0	0
Input Reg.	31	50	0	0
Output Reg.	139	76	0	0
PE	3028	3766	131	29
Total	3416	4025	131	29
Total (10 PEs)[c]	30668	37919	1310	290

[a] Adaptive Logic Module (ALM).
[b] DSP block elements distributed on the FPGA fabric.
[c] Total resources with 10 PEs.

performance than all architectures presented in Table 4. P^{2}IP has shown a performance enhancement from 1.7 to 3.18 times faster than the state-of-the-art related works. In terms of latency, it is clear that P^{2}IP has a big advantage over the architectures that have addressed this specification.

To obtain this comparison, we have taken applications and parameters (image size) presented in [21, 14, 22, 13, 31] and similarly mapped them onto the P^{2}IP. This gave us a fair basis to compare their results against the results obtained with the P^{2}IP. As presented earlier, our proposed architecture supports any image size up to 4095×4095 pixels, only limited by the size of the line buffers present in the PEs.

The throughput of the P^{2}IP is highly dependent on the F_{max} obtained during synthesis. The obtained F_{max} will certainly be different depending on the technology where the architecture is implemented (e.g., different FPGAs vendors or families and different ASIC technology). The results presented in this section attempt to illustrate the P^{2}IP performance on a relatively old FPGA technology (40 nm process technology) which we have had available for prototyping.

6. Conclusions

In this paper, we have presented P^{2}IP, a novel coarse-grained reconfigurable systolic array for real-time image and video processing. With a run-time reconfigurable datapath associated to a dedicated programming mechanism, the P^{2}IP architecture can perform many low-level image processing

Table 4: Throughput (T) and operating frequency (F) comparison with the state-of-the-art architectures w.r.t. P^{2}IP (10 PEs) @ 268 *MHz*.

					P^{2}IP
		Image Size	F	T	T
Architecture	Application	($M \times N$ *pixels*)	(MHz)	(fps)	(fps)
CSX700 [21]	HCD[a]	720×1280	250	91	290
Diet SODA [14]	2D Filter	2720×4072	400/50[b]	13	24
CRISP [22]	2D Filter	2720×4072	115	9	24
GPU [13]	CED[c]	3936×3936	575	10	17
Hybrid [31]	CED	1472×1760	2400/1296[d]	47	103

[a] Harris Corner Detection.
[b] 400 *MHz* is the frequency of the memory and controller systems and 50 *MHz* is the SIMD frequency.
[c] Canny Edge Detection.
[d] 2400 *MHz* is the CPU frequency and 1296 *MHz* is the GPU frequency.

applications.

In our study, three image processing algorithms were mapped on the proposed architecture in order to validate it. Reliable output results were achieved and discussed for that purpose. We also show how each algorithm is mapped on the architecture via the P^{2}IP configuration mechanism on different PEs all configured to become one application. For different image sizes, we show the latency of our proposed architecture and compared to that of the state-of-the-art architectures, presenting very competitive performances with a throughput of one processed pixel per clock cycle independently of the operating frequency. These obtained results proved that the proposed architecture can be a suitable low-level image processor candidate for commercial embedded systems targeting modern video standards. Finally, from a System-on-Chip point of view, the P^{2}IP can be seen as a new flexible building block with an industrial standard interface base on the AXI4-Stream interconnection protocol.

Regarding future extensions of the present work, an important addition could be an evolution of the MATLAB-based function library by using higher-level entry methods such as MATLAB/Simulink or LabVIEW. Furthermore, the addition of an expert system capable of inferring routing solutions and adjusting internal delays based on a desired P^{2}IP configuration could be very beneficial to the architecture acceptance. This expert system could

make the internal routing and delay balancing transparent for the user and complemented by a higher level entry method.

References

[1] T. R. Savarimuthu, A. Kjær-Nielsen, A. S. Sørensen, Real-time medical video processing, enabled by hardware accelerated correlations, Journal of Real-Time Image Processing 6 (2011) 187–197. doi:10.1007/s11554-010-0185-2.

[2] M. Gorgoń, Parallel performance of the fine-grain pipeline FPGA image processing system, Opto-Electronics Review 20 (2012) 153–158. doi:10.2478/s11772-012-0021-2.

[3] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, A. Das, Evaluating the Imagine stream architecture, in: Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA04), 2004.

[4] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, J. D. Owens, Programmable stream processors, Computer (2003) 54–61.

[5] IEEE, IEEE Standard Glossary of Computer Hardware Terminology, IEEE Computer Society, 1994.

[6] A. P. Reeves, Parallel computer architectures for image processing, Computer Vision, Graphics, and Image Processing 25 (1) (1984) 68–88. doi:10.1016/0734-189X(84)90049-5.

[7] K. Diefendorff, P. K. Dubey, How multimedia workloads will change processor design, Computer (1997) 43–45.

[8] B. H. McCormick, The Illinois pattern recognition computer-ILLIAC III, IEEE Transactions on Electronic Computers EC-12 (6) (1963) 791–813. doi:10.1109/PGEC.1963.263562.

[9] B. K. Khailany, T. Williams, J. Lin, E. P. Long, M. Rygh, D. W. Tovey, W. J. Dally, A programmable 512 GOPS stream processor for signal, image, and video processing, IEEE Journal of Solid-state Circuits 43 (1) (2008) 202–213.

[10] C. Bobda, R. Hartenstein, Introduction to reconfigurable computing : architectures, algorithms, and applications, Springer-Verlag, Dordrecht, 2007.

[11] D. Baumgartner, P. Roessler, W. Kubinger, C. Zinner, K. Ambrosch, Benchmarks of low-level vision algorithms for DSP, FPGA, and mobile PC processors, in: B. Kisacanin, S. Bhattacharyya, S. Chai (Eds.), Embedded Computer Vision, Advances in Pattern Recognition, Springer London, 2009, pp. 101–120. doi:10.1007/978-1-84800-304-05.

[12] K. Illgner, DSPs for image and video processing, Signal Processing 80 (11) (2000) 2323 – 2336, Special section on DSP in Audio-visual communications. doi:10.1016/S0165-1684(00)00120-1.

[13] Y. Luo, R. Duraiswami, Canny edge detection on NVIDIA CUDA, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2008, pp. 1–8. doi:10.1109/CVPRW.2008.4563088.

[14] S. Seo, R. G. Dreslinski, M. Woh, C. Chakrabarti, S. Mahlke, T. Mudge, Diet SODA: a power-efficient processor for digital cameras, in: Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design, ISLPED '10, ACM, New York, NY, USA, 2010, pp. 79–84. doi:10.1145/1840845.1840862.

[15] S. Vassiliadis, D. Soudris (Eds.), Fine- and Coarse-Grain Reconfigurable Computing, Springer, 2007.

[16] S. Hauck, A. DeHon (Eds.), Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[17] N. S. Voros, A. Rosti, M. Hbner, Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach, Vol. 40 of Lecture Notes in Electrical Engineering, Springer Netherlands, 2009. doi:10.1007/978-90-481-2427-5.

[18] J. M. P. Cardoso, M. Hübner (Eds.), Reconfigurable Computing: From FPGAs to Hardware/Software Codesign, Springer, 2011. doi:10.1007/978-1-4614-0061-5.

[19] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, R. R. Taylor, PipeRench: A virtualized programmable datapath in 0.18 micron technology, in: IEEE Custom Integrated Circuits Conference (CICC), 2002.

[20] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, The Raw microprocessor: A computational fabric for software circuits and general purpose programs, IEEE Micro (2002) 25–35.

[21] A. Fijany, F. Hosseini, Image processing applications on a low power highly parallel SIMD architecture, in: IEEE Aerospace Conference, 2011, pp. 1–12. doi:10.1109/AERO.2011.5747456.

[22] J. Chen, S.-Y. Chien, CRISP: Coarse-grained reconfigurable image stream processor for digital still cameras and camcorders, IEEE Transactions on Circuits and Systems for Video Technology 18 (9) (2008) 1223–1236. doi:10.1109/TCSVT.2008.928529.

[23] S. Mahmoudi, P. Manneback, C. Augonnet, S. Thibault, Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes, in: 20ème Rencontres francophones du parallélisme (RenPar'20), 2011.

[24] ARM, AMBA open specifications (Jan. 2013).
URL `www.arm.com`

[25] Xilinx, Hierarchical design methodology guide (Mar. 2011).
URL `www.xilinx.com`

[26] Altera, Quartus II Handbook v12.1 (Nov. 2012).
URL `www.altera.com`

[27] C. Poynton, Digital Video and HDTV Algorithms and Interfaces, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[28] C. Harris, M. Stephens, A combined corner and edge detection, in: Proceedings of The Fourth Alvey Vision Conference, 1988, pp. 147–151.

[29] T. B. Moeslund, Introduction to Video and Image Processing, Springer, 2012.

[30] W. Burger, M. J. Burge, Digital Image Processing: An Algorithmic Introduction using Java, Texts in Computer Science, Springer London, 2008.

[31] F. Lecron, S. A. Mahmoudi, M. Benjelloun, S. Mahmoudi, P. Manneback, Heterogeneous computing for vertebra detection and segmentation in X-ray images, International Journal of Biomedical Imaging Volume 2011 (2011) pp. 1–12. doi:10.1155/2011/640208.

Paulo Possa received his B.Sc. degree in Electronics Engineering from the University of Passo Fundo, Brazil, in 2005. In 2008, he received his M.Sc. degree in Biomedical Engineering from the Federal University of Santa Catarina, Brazil. He received his Ph.D. degree in Engineering Science from the University of Mons, Belgium, in 2013. His main research interests are reconfigurable computing, real-time embedded systems, digital image processing, and HDL and hardware design.

Naim Harb is a senior researcher for the Electronics and Microelectronics Department of the University of Mons, Belgium. He received his BSc degree in Computer and Communication Engineering from the Islamic University of Lebanon, Lebanon, in 2005. In 2008, he received his Engineering Master's degree in Electrical and Computer Engineering from the American University of Beirut, Lebanon. He obtained a Ph.D. degree in Computer Science from the University of Valenciennes, France, in 2011. His research interests are embedded systems, FPGAs, partial and dynamic reconfiguration, image and signal processing, bioinformatics, hardware system design and optimization.

Eva Dokládalová received her Ph.D. degree in Mathematical Morphology at the Ecole des Mines de Paris (currently Mines-ParisTech) in 2004. After the thesis, she joined the Laboratory of Embedded Computers and Image at the Atomic Energy Commission in Saclay, where she contributed to research projects in the field of embedded architectures for image and video compression. Currently, she is a Professor-Researcher at the Computer Science Department of the ESIEE-Paris. Her research interests are focused on hardware architectures for advanced real-time embedded vision systems and image processing algorithms for embedded vision.

Carlos Valderrama is the director of the Microelectronics Department at the University of Mons, Belgium, which is dedicated to the design of integrated electronic circuits and digital systems. Previously, he was leading projects in CoWare in Belgium, and a Visiting Professor at Brazilian Universities. He received the EE engineering diploma from the National University of Cordoba, Argentina, in 1989. In 1993, he received the MSc in Microelectronics from the Federal University of Rio de Janeiro, Brazil. He obtained his Ph.D. degree from the Grenoble Institute of Technology, France, in 1998.