Microprocessors and Microsystems 39 (2015) 576–588

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Processor arrays generation for matrix algorithms used in embedded platforms implemented on FPGAs

Roberto Pérez-Andrade [a,*], César Torres-Huitzil [a], René Cumplido [b]

[a] Information Technology Laboratory, Advanced Studies Center of the National Polytechnic Institute, CINVESTAV, Ciudad Victoria, Mexico
[b] Department of Computer Science, National Institute for Astrophysics, Optics, and Electronics, INAOE, Santa Maria Tonantzintla, Puebla, Mexico

ARTICLE INFO

Article history:
Received 1 July 2014
Accepted 23 December 2014
Available online 10 February 2015

Keywords:
Processor arrays
Polytope
Embedded platforms
FPGAs

ABSTRACT

Matrix algorithms are an important part of many digital signal processing applications as they are core kernels that are usually required to be applied many times while computing different tasks. Hardware assisted implementations using FPGAs provide a good compromise between performance, cost and power consumption, specially when high level synthesis techniques are employed for deriving co-processors. In this paper a high level synthesis approach to generate embedded processor arrays for matrix algorithms based on the polytope model is presented. The proposed approach provides a solution for efficient data memory accesses and data transferring for feeding the processor array, as well as support for solving problems independently of their size and limited only by the FPGA available resources. The proposed approach has been validated by generating processor arrays for three different matrix algorithms used in digital signal processing applications; more precisely matrix–matrix multiplication, Cholesky and LU decomposition algorithms. These algorithms were targeted for a Spartan-6 device and compared against their sequential implementations targeted for a MicroBlaze processor in order to provide a general view of the gain achieved by the processor arrays when the arrays and sequential processors are implemented in the same technology. Results show that the implemented arrays outperforms hardware and software implementations considering an embedded platforms scenario with a Spartan-6 device.

1. Introduction

Traditionally, the intensive computational performance required in several applications were met only by technology advances like smaller transistor area, higher clock rates and other improvements. However, problems like power dissipation and thermal constraints have emerged as dominant design issues; forcing computer designers away from relying on increasing frequency to improve performance. Thus, using multiple processing units for performing parallel computations and completing a larger volume of work in shorter time periods has became the current trend in order to improve performance in several domains, like digital signal processing [1]. Nowadays, the main trend for achieving greater performance and increasing the computational efficiency is by exploiting different forms of parallelism such as instruction level parallelism (ILP), data level parallelism (DLP) or loop level parallelism (LLP).

Among these forms of parallelism, the LLP approach is widely used in the scientific computing and digital signal processing communities, since roughly at least an 80% of the execution time of sequential programs is typically spent in computing nested loops, which represent the 20% of program codes [2]. Besides, several numeric kernels used in digital signal processing applications can be computed in forms of nested loops, representing niches of opportunities for being implemented in hardware platforms exploiting the LLP. These numeric kernels are required in order to build "complex" algorithms on electronic systems. Algorithms such as matrix multiplication, matrix decomposition, convolution and system equations solvers are used as base for building more complex systems. For example, matrix multiplication is required in some Fast Fourier Transform (FFT) algorithm implementations [3]; whereas matrix decompositions, such as Cholesky, LU or QR, are used in MIMO (Multiple Input Multiple Output) systems [4], facial feature extraction [5], classification and target tracking in wireless sensor networks [6], orthogonal matching pursuit for signal reconstruction in compressed sensing theory [7], and least-square estimation (LSE) and multiple-parameter linear regression (MLR) [8].

* Corresponding author.
E-mail addresses: jrperez@tamps.cinvestav.mx (R. Pérez-Andrade), ctorres@tamps.cinvestav.mx (C. Torres-Huitzil), rcumplido@ccc.inaoep.mx (R. Cumplido).

http://dx.doi.org/10.1016/j.micpro.2014.12.003
0141-9331/

Traditionally, the parallelism of the aforementioned numeric kernels has been exploited by using digital signal processors (DSPs) which are optimized for performing in parallel the most common used operations in signal processing applications like multiplication–accumulation (MAC) operation, and in recent years highly specialized coprocessor units based on field programmable gate arrays (FPGAs) platforms have been used for exploiting the LLP by taking advantage of spatial computing paradigm [9]. However, a considerable effort and time are required to develop such hardware implementations leaving to computer designers the responsibility of crafting these highly specialized application-specific architectures. Hence, hardware assisted approaches for automatically generating dedicated hardware architectures, are beneficial for implementing loop-based algorithms and extracting their parallelism.

High-level synthesis (HLS) methods allow the automatic generation of hardware circuits from behavioral descriptions, favoring a faster exploration and evaluation of possible hardware architectures compared to traditional hardware design flows. Generally, HLS methods try to extract automatically the parallelism from an algorithmic representation, and at the same time, derive parallel hardware structures from this algorithmic input. The target hardware circuit derived by HLS methods consist of a structural composition of data-path, control and memory elements. The fundamental tasks in HLS methods are decomposed into hardware modeling, scheduling, resource allocation and binding, and control generation [10]. Although several HLS methods can be found in the literature, each of them uses different hardware models deriving different kinds of hardware architectures. Models like the control-data flow graphs (CDFG) [11], the hierarchical task graphs (HTGs) [12], the Kahn process networks [13] and the polytope model [14] have been employed by the HLS community. Although some these HLS approaches use as algorithmic representation loop-based algorithms they do not offer any parallelization, since they generate highly-pipelined mono-processors in order to achieve a higher data throughput. However, the polytope model [10] is able to expose the loop level parallelism of sequential loop-based programs by providing an abstraction to represent loop computations of an input specification as integer points inside of a polyhedron. As a result, the polytope model could be used for the synthesis of hardware architectures exploiting the LLP in digital signal processing algorithms in the form of highly-pipelined mono-processors [15] or processor arrays [16,17]. Processor arrays are regular, locally connected and massively parallel architectures with simple processing elements (PEs), whose structure is well suited for their implementation into FPGAs.

In this paper, a HLS approach for generating processor arrays for matrix based algorithms using the polytope model is presented. The proposed approach provides a complete system, integrating the processor array data-path (PEs architecture and processor array interconnection), the control structure for generating the activation and control signals, and the memory interface for inserting/extracting data to/from the processor array which could be used in FPGAs or in VLSI designs. The rest of the paper is organized as follows: Section 2 presents a general overview of related works concerning the processor array generation within the polytope model context and a brief discussion about these works. A description of the processor array generation using the polytope model is presented in Section 3. Section 4 describes the proposed architectural designs focusing on the controller generation and the memory system for inserting/extracting data to/from the processor array. Resource utilization, acceleration and throughput results compared against a soft-processor implementation for three algorithms are presented in Section 5, while the conclusions are presented in Section 6.

2. Related work

Although several HLS methods have been proposed in the literature, only few of them are focused on the generation of parallel architectures by using the polytope model. In this section a review of these HLS methods is presented. One of these works about HLS within the polytope scope is the MMAlpha programming environment [17], which transforms an input specification in form of a system of uniform recurrence equations (SURE) into VHDL code describing a processor array, and to a C code for simulation purpose. MMAlpha has been targeted for solving linear equation systems, string matching, computing scores for hidden Markov model, finite impulse response (FIR) filter and matrix–matrix multiplication (MatMul). One characteristic that these algorithms share is that their loop bounds form rectangular shapes. In fact, cases where the design methodologyfollowed by MMAlpha is used for generating processor arrays (full-size or partitioned) for algorithms whose loop bounds form non-rectangular shapes (*e.g.* back substitution, Cholesky, QR and LU decompositions algorithms) have been not shown. Moreover, although all processor arrays generated in a full-size fashion are size-dependent, processor arrays synthesized by MMAlpha using partitioning transformation are unable to solve several problem size instances, *i.e.* if the problem size for which the processor array was derived changes, a new processor array with its respective controller should be generated. Within the context of MMAlpha environmental tool, Plesco in [18] presents a hand-made solution for interfacing an external memory with a processor array of 4×4 PEs generated by MMAlpha tool by using the MatMul of complex numbers of 32-bit word size. This hand-made solution is only a specific implementation without any generalization for other cases of study. Also, within the MMAlpha framework and using the MatMul algorithm, Derrien in [19] deals with I/O aspects involved in the processor array generation by proposing a methodology to derive a set of conflict free I/O data pipelines along the processor array boundaries.

Another tool for automatic processor array generation is PARO [16], which is able to map computational intensive nested loop programs into parallel architectures that are translated into VHDL code. Similarly to MMAlpha, PARO uses a special input specification in form of SURE called dynamic piecewise regular algorithms (DRPAs), which are a generalization of the SUREs. The PARO processor array synthesis consists of three steps: synthesis of the processor elements, generation of the control structure and derivation of the interconnection topology [20]. The hardware synthesis generates a register transfer level (RTL) description which is later translated and optimized into VHDL code. Also, PARO is able to automatically generate VHDL test-benches in order to perform the verification of the processor array. The controllers generated by PARO use combined global and local control facilities. Such controllers are in charge of orchestrating data transfer and computations for processor arrays, and they are based on the use of counters, decoders, address generators, and glue logic for interfacing the processor array to other components integrated in system-on-a-chip (SoC) environments. However, the data I/O is only proposed to be done either by functional simulation, by direct memory access (DMA), or by software running on a host processor [21]. Moreover, cases of study presented during the development of PARO tool include MatMul algorithm [22], FIR filter [23], discrete cosine transform (DCT), and images filters like edge detection, bilateral [24] and gaussian filters [16]. MatMul, FIR and edge detection algorithms are cases of study that show how PARO could be used for generating partitioned arrays [16]. As the same case of MMAlpha, algorithms targeted in PARO tool have the rectangular loop bound shape characteristic. Although during the development of PARO, the mathematical theory for deriving processor arrays

from algorithms with non-rectangular loop bound shapes has been developed, cases of study where partitioning transformations are applied to algorithms whose loop bounds are not rectangular (like QR, Cholesky an LU decomposition algorithms) have been not shown. Moreover, processor arrays derived by PARO are size dependent due to the assumption of fixed tile sizes [24], and they do not use complex hardware operations used in matrix decompositional algorithm.

Another framework for mapping perfectly nested loops into processor arrays implemented in FPGAs is presented by Uday et al. in [25]. In this framework, two global controllers associated with different dimensional times are used for the control signal generation, and each one of these controllers streams the activation signals from a particular corner of the processor array with certain delay. Partitioning is supported by using multidimensional scheduler in a local parallel global serial approach, but it lacks of information about any memory support (like FIFO elements) for the used partitioning approach. The authors show only the MatMul algorithm implemented as a processor array, but information about an external memory module in charge of providing data is not mentioned.

An automatic tool for the generation of processor arrays is the PICO-NPA (Program-In Chip-Out and Non-Programmable Accelerator), which is a commercial system for synthesizing hardware co-processors from perfectly nested loop C programs [26]. The project was developed in the HP Palo Alto research laboratory and now it is being commercialized by Synopsys with the name of PICO Express [27]. This tool generates synthesizable HDL code that defines several RTL algorithmic specifications and HDL test-bench codes for each RTL specification. PICO is able to exploit four different levels of parallelism: loop, instruction, inter-task and intra-task. Basically, the PICO target architecture is built on three hierarchical levels. The first one consists of simple processing elements containing arithmetical units. The second level consists of a set of processing elements (PEs) locally interconnected called processing array (PA), which incorporates local memories for reducing the bandwidth of external memory accesses. The third is made of a set of PAs connected by FIFO memories called pipeline of processing arrays (PPA). A controller is in charge of orchestrating the operations of all PAs, while an interface is used to communicate with a host computer. Data transfers from CPU to the PPA are realized by specialized hardware units. It should be noted that a PA is similar to the processor arrays in the sense that a it is composed by simple processing elements interconnected between them in a regular and local fashion. Unfortunately, since PICO is a proprietary technology, several details are omitted.

One recent HLS work for deriving sequential hardware with a high parallelism level is proposed in [28] by Alias et al. This methodology relies on FloPoCo an open-source tool for FPGA floating point arithmetic-core generation [29]. As input specification it takes a C program and the amount of desired pipeline stages, producing VHDL code as result. The input C program is changed by using some transformation within the polytope context like the loop blocking transformation. Alias et al. show that, from this sequential implementation, obtaining a parallel version by replicating the sequential processor is possible. The control generated by their semi-automatic methodology is composed by a single finite state machine (FSM) that captures the whole loop nest execution sequence. Also, this control generates external memory addresses for where the input/output data is stored.

Although tools like PARO and MMAlpha produce partitioned processor array in different ways, they lack of parametric support. If the problem size for which they were targeted changes, theactivation sequence changes too, and consequently a new processor array must be synthesized in order to generate a new activation sequence. Besides, the cases of study used in the aforementioned works have been focused on algorithms whose loop bounds form rectangular shapes, leaving behind decompositional matrix algorithms like QR, LU and Cholesky, which could be used in embedded platforms. It is important to emphasize that albeit PARO provides mathematical support for any kind of loop bound shapes, it does not show any case of study of processor arrays implementation with non-rectangular loop bounds. Additionally, there are few attempts for deriving memory interfaces for processor arrays constructed by using the polytope model, and most of them are limited to specific algorithms like in [18] without any generalization for the other cases of study. In these reviewed works, it is assumed that data are fed and read to/from the processor array as needed and no details are given about the external memory interface.

Table 1 shows a detailed comparison among the three most similar works related to this work. This table specifies the kind of algorithm that each tool has been targeted, the kind of allocation technique supported by the tool and the three main characteristics of this work compared to other ones: the support for non-rectangular loop bound shape, problem size independency and external memory support. Mainly, the allocation techniques consist of four different approaches. The projection approach results on full-size processor arrays dependent of the problem size, whereas co-partitioning, local parallel-global serial (LPGS) and local serial-global parallel (LSGP) approaches derive processor arrays which virtualize the full-size array into a smaller array with a fixed number of processors.

3. Processor array generation on the polytope model

The polytope model provides an abstraction to represent computations in a sequential loop program or in a more general representation like a piecewise linear algorithm (PRA), which is a specific form of the SUREs. A PRA is a set of N quantified equations and each equation $Si[I]$ is defined for all $I \in \mathcal{I}_i$ according to:

$$x_i[P_i \vec{I}] = \mathcal{F}_i(\ldots, x_j[Q_j \vec{I} - \vec{d}_{ji}, \ldots]) \quad \text{if } C_i^l(\vec{I}) \tag{1}$$

where x_i, x_j are affinely indexed variables. The indexing functions of the variables are defined by the constant indexing identity matrices Pi, Qj and by the i-th constant integer *dependence vector dji* of the corresponding dimension. $Ci^l(\vec{I})$ is called *iteration dependent* condition of an equation. $\mathcal{F}^i$ denotes arbitrary functions and the dots denote similar arguments. Ii is an integral subset $\mathbf{I} \subseteq \mathbb{Z}^n$ called *iteration space* of the PRA. The vector I represents an *iteration point* $I \in \mathcal{I}_i$. Some variables in the PRA represent the data input and output from an arbitrary external source in form of *I/O variables*.

A general design flow followed for the generation of processor arrays within the polytope model is shown in Fig. 1a. First, the original program, or *source polytope* is represented as a PRA (Fig. 1b). From the source polytope, information like data dependences (in form of dependence vectors d_{ji}) among the variables, latency of operations in hardware, and the type of variable present in the PRA is extracted and gathered into a *reduced dependence graph (RDG)*. Together, the iteration space $\mathcal{I}$ and the RDG are employed to define a *scheduler* ($\vec{\lambda}$) and an *allocation* (Φ) functions. Basically, the purpose of the scheduler is to assign a computation date for each task of the PRA; whereas the allocation assigns the tasks to all the PEs such as no two tasks with the same computation date are assigned to be executed on the same PE. In the context of this work, the linear scheduling proposed by Darte et al. in [30] is used, since it derives asymptotically functions equivalent to the best scheduler; that is the difference between the total execution time of the best linear scheduler and a scheduler which assign computation times as soon as data are available is bounded by a constant independent of the problem size. On the other hand, the allocation

Table 1
Comparison of the proposed approach against other works.

	PARO [16]	MMAlpha [17]	Uday [25]	This work
Algorithms supported				
Matrix–matrix based	✓	✓	✓	✓
FIR and convolution	✓	✓	✗	✓
Image filtering	✓	✗	✗	✗
Matrix decompositions	✗	✗	✗	✓
Allocation technique				
Projection	✓	✓	✗	✓
LPGS	✓	✗	✓	✓
LSGP	✓	✓	✗	✗
Co-partitioning	✓	✗	✗	✗
Non-rectangular	✗	✗	✗	✓
Problem size independency	✗	✗	✗	✓
External memory support	✗	✗	✗	✓

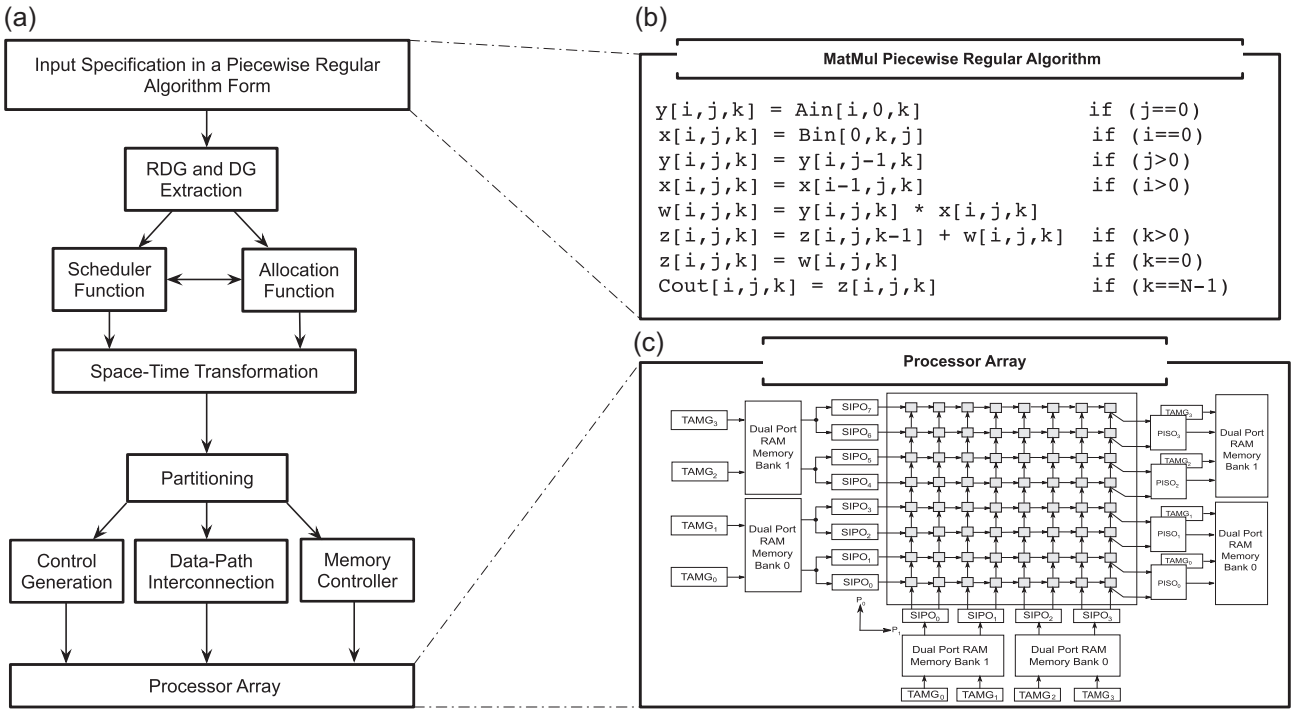

Fig. 1. (a) Processor array design flow followed in this research. (b) Matrix–matrix multiplication PRA. (c) Processor array derived by the design flow.

functions used in this work are obtained from a user proposed projection vector $\vec{u}$, according to [16]. By using the projection vector, it is possible to derive full-size processor arrays which are later partitioned. Along with the scheduler and the allocation function, a time interval between two successive computations performed by the same PE, called *iteration interval*, is calculated from these functions. Together, the scheduler and allocation functions are used to perform a space–time mapping over the source polytope in order to obtain the *target polytope*. The target polytope contains the same iteration points of the source polytope, but in a new coordinate system in which a dimension is strictly temporal and the others are strictly spatial, *i.e.* the space–time mapping divides the source polytope $\mathcal{I}$ into two subspaces $\mathcal{T}$ and $\mathcal{P}$ which define a time and a processor space, respectively.

Applying the space–time transformation full-size problem dependent processor arrays unsuitable of being implemented for large iteration spaces can be derived. Partitioning helps to derive processor arrays independent of the problem size by using a fixed number of processors and preserving the interconnection topology among PEs. Depending on the interpretation of the partitioning transformation several approaches could be obtained. The LPGS partitioning approach refers to compute in parallel the iteration points covered by a tile and execute the remaining tiles sequentially. On the other hand, the LSGP approach refers to execute the iteration points inside of the congruent tiles sequentially, but computing the rest of tiles in parallel fashion. In this work, the processor arrays using the LPGS approach are derived by using *strip mining* technique. Strip mining consists of dividing the dimension of the processor space into strips, resulting in processor spaces divided by congruent *tiles*. Each strip divides one dimension of the processor space by a constant stride of size *SSpk*, and it adds new dimensions for scanning them without adding these indexes to the PRA [31]. The stride size *SSp0* defines the size of a one-dimensional processor array, whereas *SSp0* and *SSp1* define the size of a two-dimensional array, *i.e.* $\mathcal{T} \in \mathbb{Z}$ and $\mathcal{P} \in \mathbb{Z}^2$, respectively. Once these transformations (space–time mapping and partitioning) have been applied, the controller [16,17], the processor array topology [20,24], the PE data-path [16,17] and the memory

controller [18,32] are synthesized (Fig. 1c). The controller indicates when a PE inside the processor array must be activated at a given time, and which operation must be performed inside of the PE. The interconnection topology specifies how data travels through the array, while the PE data-path performs the computations required by the algorithm. Since the processor space is partitioned, some FIFOs elements are added at the processor array borders in order to store intermediate data produced by the array when the processor space is being scanned by the tile indexes, avoiding external memory accesses. Finally the memory system feeds the processor array with data coming from an external memory, and at the same time, it extracts the data results produced by the array storing them into an external memory. In the following section, the generation of the processor array following the design flow shown in Fig. 1a. is explained.

4. Architectural design

The proposed architectural design consist of three major blocks: the processor array data-path, the controller, and the memory interface. It is important to emphasize that for deriving the processor array data-path of the proposed architecture well-known mathematical expressions are used [16,17]. However, for sake of completeness, a brief overview of the derivation of PE data-path and the interconnection topology of the PEs is presented in Section 4.1. Sections 4.2 and 4.3 are advocated to describe the proposed controller and memory interface.

4.1. Data-path

By using the dependence vectors, scheduler, allocation, and RDG it is possible to determine if a local connection among processing elements exists, the number of delay elements (or registers) that must be placed between each connection as well as the internal PE data-path. Basically, the interconnection of the PEs is obtained by using the allocation function, and the data dependences different of zero. Intuitively, for each data dependence vector $\vec{d}_{ji} \neq 0$ in the PRA, a connection from the indexed variable x_j to x_i is inferred, *i.e.* the number of dependence vectors different from zero provides the number of PE input/output ports. Each interconnection $s_{ji} \in \mathbb{Z}$ corresponds with a dependence vector d_{ji} in the PRA. If s_{ji} is different from zero, it indicates that a connection should be placed between the indexed variables j and i, otherwise a PE internal feedback should be placed.

The value of s_{ji} is calculated as follows:

$$S_{ji} = \Phi \vec{d}_{ji}$$

On the other hand, the number of registers required between each interconnection s_{ji} is determined by using the scheduler function and data dependence vector $\vec{d}_{ji}$. Similarly to the interconnection, a delay equal to r_{ji} is set between the indexed variables xj and xi according to:

$$S_{ji} = \vec{\lambda} \vec{d}_{ji}$$

Finally, the derivation of the PEs data-path is accomplished by binding the RDG nodes to hardware elements according to $C_I(\vec{I})$, resulting on different types of PEs inside of the processor array. For example, after space–time transformation, only some PEs will perform external I/O memory communication while other PEs will perform a subset of the PRA operations. Thus, the data-path for each processor type is synthesized from the RDG, since each one of its nodes denotes a different kind of functionality (input, output, propagation, and operational). The operational nodes are directly bound to hardware functional units like adders, multipliers,and dividers. Some multiplexers are added in case of there are several nodes corresponding to the same indexed variable. The RDG nodes labeled as input or output denote I/O ports of the processor array. These ports are in charge of receiving all data from an external source and of sending data results to an external source. It is important to differentiate these processor array I/O ports, which communicate the processor array with the memory interface, from the PE I/O ports which are used for interconnect the PEs.

4.2. Controller

The processor array controller is based on a centralized and distributed modules (Fig. 2). In this approach, the most costly hardware and repetitive operations are placed in the central modules in order to reduce possible overhead introduced if such operations were placed in a distributed way; whereas simple operations are placed into the distributed modules. In this sense, the centralized control module generates the scanning order of the tile and time indexes after applying strip mining; whereas the distributed modules propagates such indexes as well as the control signals through the processor array; and if it is required, these signals are modified by the distributed modules.

The centralized part is composed by two modules called *sequence generator* and *activation-signal injector*; whereas the distributed part is made of several control cells forming a *control array* whose interconnection topology is identical to the PEs in the processor array. The idea behind the controller is having an special unit in charge of scanning the divided processor space $\mathcal{P}$, and generating the time index derived from strip mining and space–time mapping, respectively. Later these indexes are decoded in order to know which PE is the first to be activated at the beginning of each tile, and then propagating an activation signal through the control array as well as some other information required by the control cells (like the problem size and current tile iteration). A detailed description of this controller can be found in one previous work [31]. However, for sake of completeness, the next subsection briefly summarize the three controller modules.

The Sequence Generator is a set of specialized counters able to generate the scanning order of the tile and time indexes obtained after applying strip mining $\mathcal{P}$. This module is composed by a set counter-like submodules connected in a cascade fashion. Each counter-like submodule is able to count between different ranges of values according to the limits calculated by a combinational Max/Min submodule. This submodule calculates the maximum and minimum values at execution time. This kind of run-time evaluation, required for supporting a set of problem sizes, is one of the parameters required by the expressions mapped to the Max/Min submodules. Inside of each counter-like submodule there is an internal counter which enables the counting of the counter-like module at a rate equal to the iteration interval obtained from the scheduler and allocation functions, *i.e.* each counter is not necessarily incremented by a unitary step. For each non-processor index (*i.e.* time and tile indexes) presented in the partitioned target polytope, a pair of a Max/Min and counter-like sub-module is required. The advantage of using combinational logic in the Max/Min submodules is that if the bounds of the partitioned processor space were changed, only by changing the Max/Min expression, the sequence generator is able to generate the new tile and time indexes. Moreover, by adding h-pair of counter-like and Max/Min submodules the functionality of h non-processor indexes can be achieved. Fig. 3 shows the interconnection of counter-like submodules with their respective Max/Min sub-modules when $h = 3$. The combinational submodules Max/Min are grey-shaded. Note that the Hold signal from the inner counter-like submodule is connected to a counter which establishes the iteration interval P. Some two-state finite state machines (FSMs) are in charge of generating the Load and Hold signals of the counter-like submodules, and the

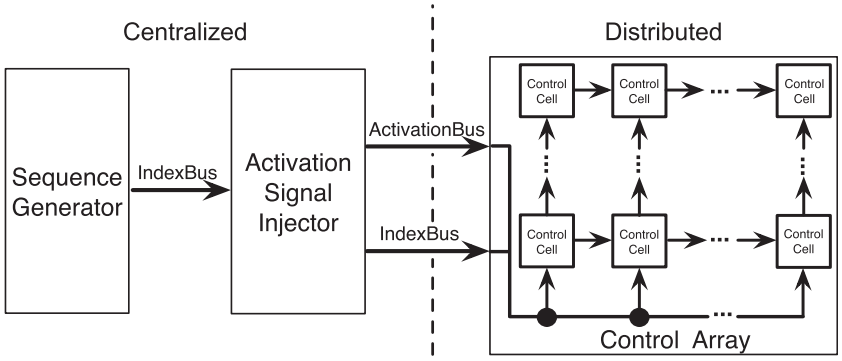

Fig. 2. Processor array controller block diagram.

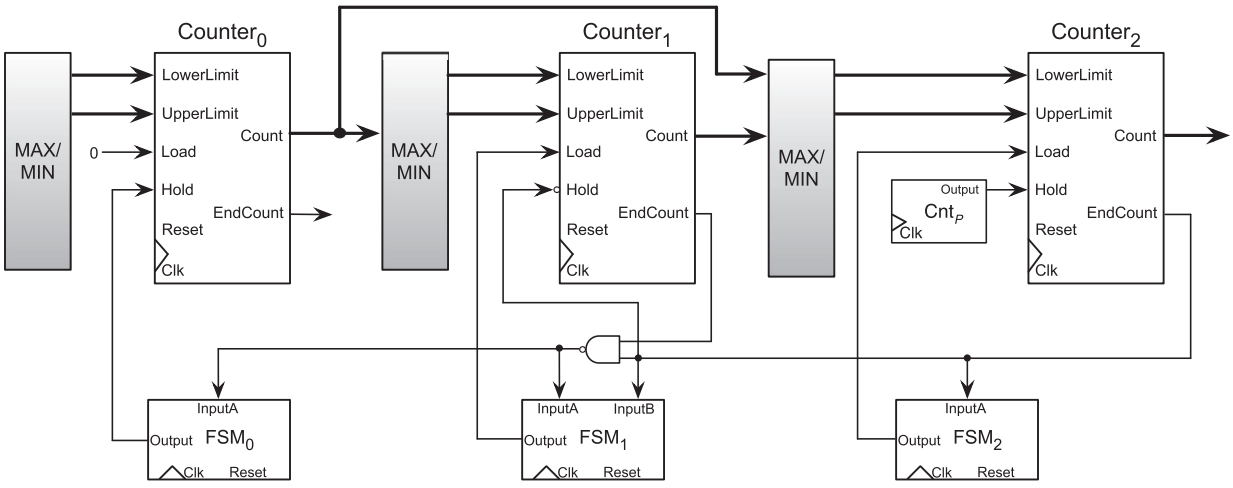

Fig. 3. Processor array controller block diagram.

transitions made by these FSM is done accordingly to the iteration interval *P*.

The Activation-Signal Injector is a combinational module in charge of selecting which PE, in the border of the processor array, must be activated at the first time instant when a new tile is being scanned. Also, it injects an index bus composed by the indexes generated by the sequence generator and by the problem size. The reason of injecting this bus is that all PEs should know what tile iteration is being executed at given time and what is the size of the problem that is being solved. This information provides to the processor array a problem size independency, making it able to compute several problem sizes, without regenerating the processor array. Thus, this module is composed by a Max and a priority decoder combinational sub-modules.

The Control Array is composed by a set of control cells, which includes combinational logic that checks the correct mapping of the processor space and some module *P*-counters for generating the activation pattern. The set of control cells is in charge of activating a subset of PEs inside of the processor array at certain time while the tiles are being scanned. Such activation occurs by circulating the activation signal and the index bus injected by the activation-signal injector module. The data circulation is performed by knowing two specific characteristics of a PRA after the space–time transformation: *the activation pattern of the PEs* and *the correct PE mapping of an iteration point* $\vec{I}$ *from the processor space* $\mathcal{P}$. The activation pattern provides an idea of how many clock cycles the incoming activation signal must be kept activated inside of control cell; whereas the correct mapping ensures that a PE maps a valid processor space index point (specially in the case of non-rectangular iteration spaces). Both characteristics are evaluated at execution time by the activation patter generator and boundaries detector units, respectively. Such run-time evaluation is helpful for generating the control signals for a set of problem sizes without the need of re-generating the controller. Thus, the activation pattern and the boundaries detector units includes combinational logic in charge of performing their respective task at execution time. Fig. 4 shows the control cell internal architecture. Coarse lines indicate the input index bus injected from the activation-signal injector, and the activation signal coming from the control cell neighbors. The activation pattern generator controls the enable signal of the corresponding PE, and it stops a FSM. Similarly to the FSMs required by the sequence generator, the state transitions of this machine are done according to the iteration interval *P*, thereby internal modulo counters are included for each control cell. The boundaries detector unit and some AND gates are in charge of deciding if the activation signal must be sent to any of neighboring cells. Moreover, the control cells have a set of registers for storing the indexes and the activation signal generated by the set of AND gates. These registers are enabled by following at the same rate as the iteration interval and respecting the scheduler.

4.3. Memory interface

The memory system consists of address generator units (AGUs), memory banks and registers working in serial-input/parallel-output (SIPO) and parallel-input/serial-output (PISO) fashion. The selection of these components, their interconnection, and their

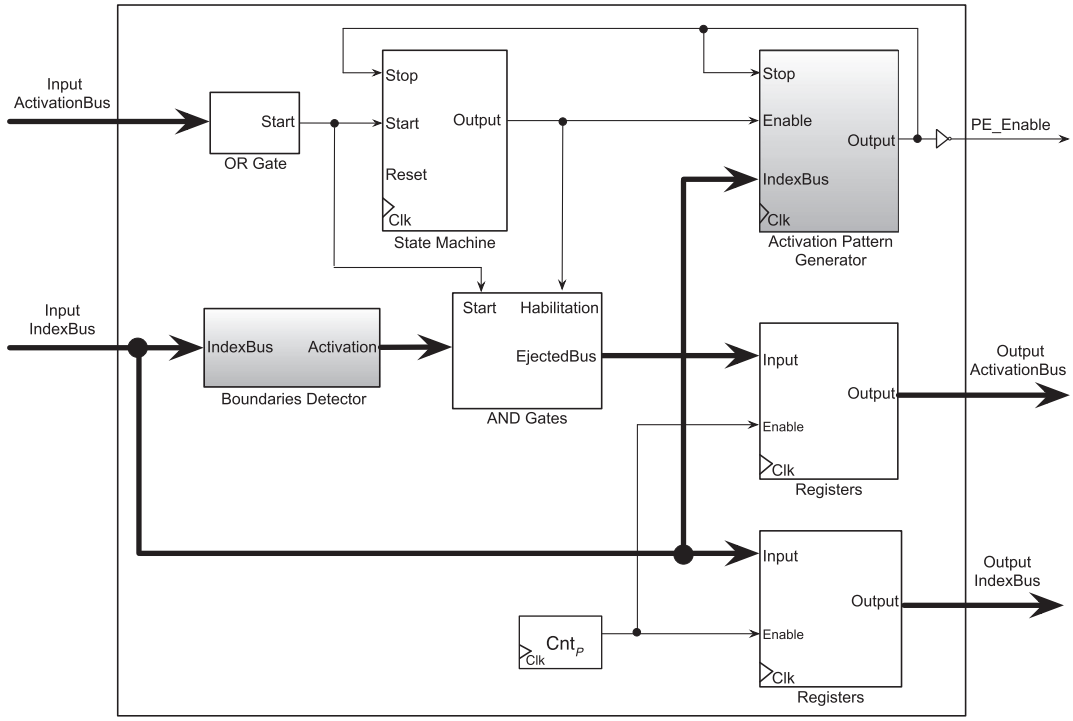

Fig. 4. Generalization of the control cell architecture.

internal architecture varies depending on the I/O variables and its iteration dependent condition after space–time. These variables can be grouped in two different possibilities: border and broadcast mapping. A border mapping occurs when the index vector $\vec{I}$ of $C_I(\vec{I})$ is transformed into processor space, and one dimension of the vector $\vec{I}$ in the I/O variable is mapped to the time space. On the other hand, a broadcast mapping occurs when the index vector $\vec{I}$ of $C_I(\vec{I})$ is transformed into time space, and all dimensions of vector $\vec{I}$ in the I/O variable are mapped to the processor space. From these two mapping possibilities, there are other two cases depending if the PRA variable represents an input or output, resulting in a total of four mapping possibilities. These I/O variables can be interpreted as a representation of external memory, with a specification of which PE will require a datum at certain time instant according to the space–time mapping. Essentially, after the mapping and depending on the variable type, four different architectural cases could be derived. These four cases assemble a memory system for inserting/extracting data to/from the processor array. These cases are: input border mapping (Fig. 5a), output border mapping (Fig. 5b), input broadcast mapping (Fig. 5c), and output broadcast mapping (Fig. 5d).

The processor array memory system satisfies the constraint that all data are required and produced by the processor array during each clock cycle respecting the algorithm data dependences. This constraint can be interpreted as the worst case scenario when the processor array is derived, *i.e.* when a clock cycle is equal to the iteration interval ($P = 1$). Also, the use of dual-port memories for each memory bank and the possibility to extract two data per memory port in a processor clock cycle are assumed. This last assumption requires to double the external memory clock frequency (*Clkmem*) with respect to the processor array clock frequency (*Clkpa*), *i.e.* $Clkmem = 2 \times Clkpa$. The combination of both assumptions leads to have a data extraction rate of four data per memory bank in a processor clock cycle. In addition, these assumptions try to provide as multiple communication channels as the processor array requires. For instance, in the case of the border mappings (either for input or output variables) the number of communication channels is given by one of the processor array size parameters, *i.e.* *SSp0* or *SSp1*; whereas in the broadcast mapping, the number o channels changes in a quadratic way as the processor array size is altered. Fortunately, all the communication channels are not used at the same time when the broadcast mapping is derived, and the original number of channels ($SSp0 \times SSp1$) could be decreased, since a fraction of these potentialcommunications are required in parallel at a same time instant. Finally, due to the four mapping cases, derived from the I/O variables and its iteration dependent condition, there are also four different architectural modules. Fig. 6 shows the internal architecture of these four cases. In these figures, for sake of simplicity, two AGUs are grouped into the TAGM sub-modules.

It should be noted that in the output border case, data produced by the processor array is recollected at the processor borders despite of they are not necessarily produced by the border PEs in the processor array. This last situation occurs when the problem size does not fit exactly in the partitioned array. In such case, data are produced by inner PEs, and these data must be sent to the processor array border. The sending of output data originated inside the array is accomplished by placing a layer of registers data array composed by a set of registers following the interconnection structure of the processor array. Another situation that should be noted is that in the input broadcast cases a block of data (equals to the number of PEs) is required each time a tile is being scanned. Such data block must be sent in advance to each PE where each datum of the data block is required. The forwarding requires sending all data contained in the block from one processor border, and sending these data simultaneously. Each time that a sub-block (a part of the data blocks) passes through a pipeline stage, one datum of the sub-block is taken by a PE while the remaining data are sent to the next pipeline stage. This approach calls for a high quantity of registers for storing the pipelined data at each pipelining stage. In fact, the set of such registers could be abstracted as an array of registers where the number of registers is decreasing as data get further from the border. This array is called broadcast data array,

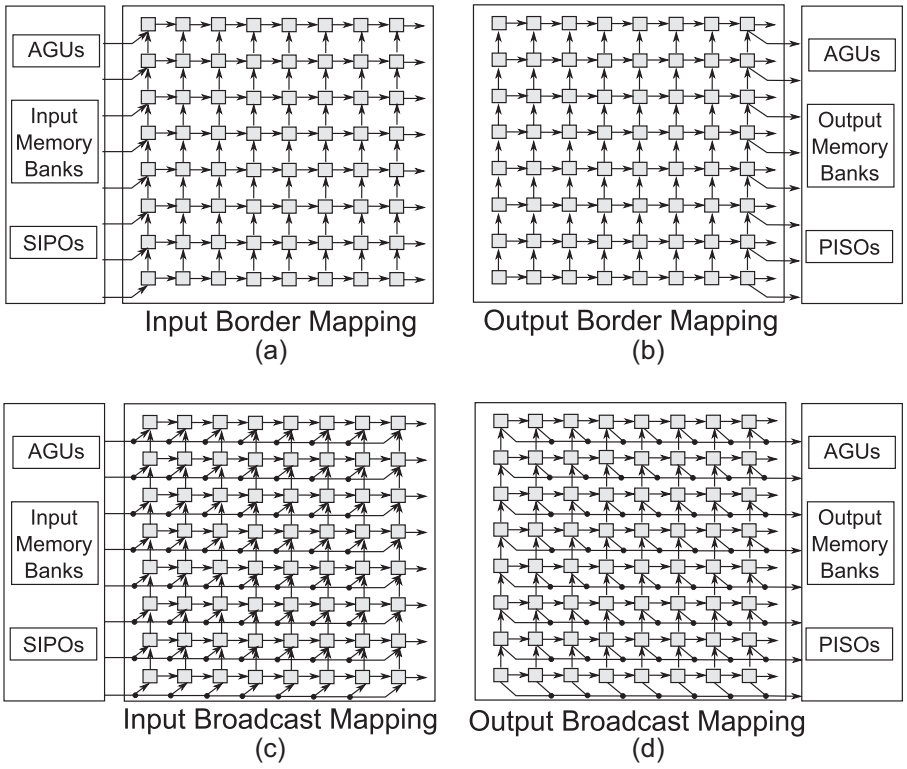

Fig. 5. Architectural cases according to the variable type and the mapping possibilities.

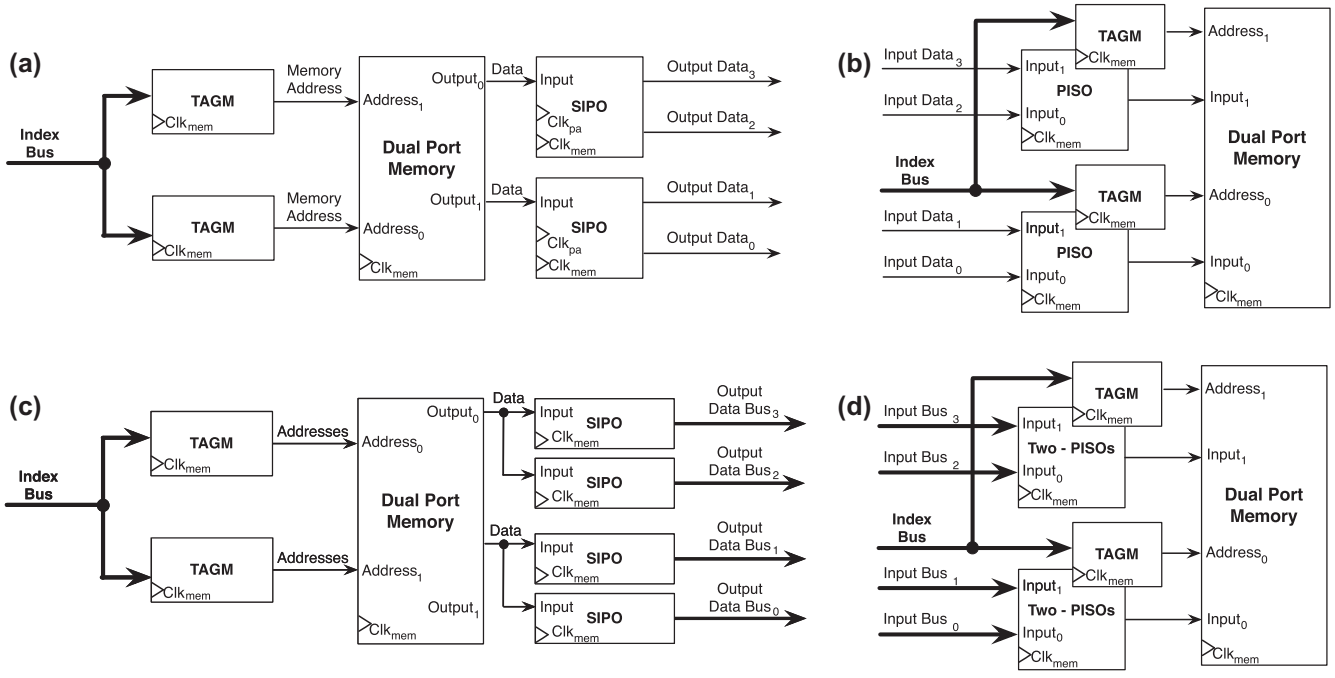

Fig. 6. Interconnection example of the (a) input border case, (b) output border case, (c) input broadcast case and (d) output broadcast case.

and an example of such array is shown in Fig. 7 for a processor array of 8×8 PEs. In this figure grey boxes represent the set of registers and the numbers above the lines indicate the amount of data that it is being pipelined. In Fig. 7, it is assumed that data travels horizontally, but this idea could apply in the case of data are propagated vertically. Besides, this same array is required in the case of output broadcast.

4.4. Integrated system

Fig. 8 shows the block diagram integrating the processor array data-path, the controller, and the memory system showing in different colors the two clock domains. In this integrated system, the tile and time indexes are generated by the sequence generator, later these indexes are decoded by the activation-signal injector

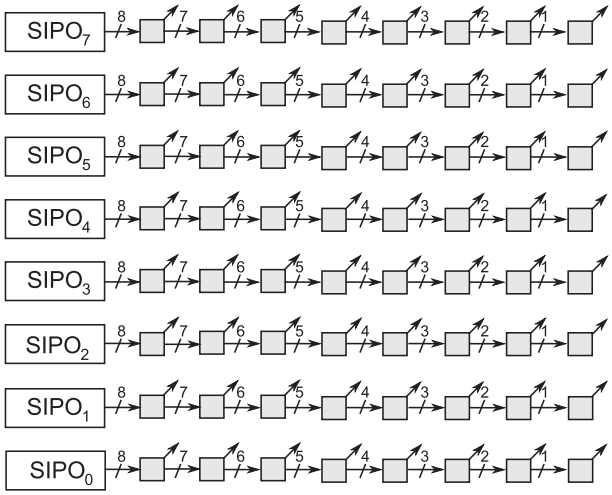

Fig. 7. Broadcast data array for an 8 × 8 processor array. From the SIPO side to the other border, the data bus is being decreased.

and by the AGUs in order to generate the processor array activation sequence and the input memory addresses, respectively. Data extracted from memory are inserted inside the processor array by the SIPO elements and at the same time the activation signal is injected to the processor array. All intermediate data produced by the array is stored in FIFOs and it is reused without accessing external memory (like a scratch memory). Once results are being produced by the processor array they are recollected by PISO registers and they are stored in an output memory. It is important to emphasize that the integrated system is able to support different scheduler functions by changing the mathematical expressions mapped to combinational logic in:

- The Max/Min sub-modules located in the sequence generator (control scheme).
- The Max sub-modules placed in the activation-signal injector (control scheme).
- The boundaries detector and the activation pattern generator sub-modules in the control cells (control scheme).
- The AGU sub-modules required by each architectural case (external memory).

Moreover, if the projection vector is also changed, by the correct selection of the memory architectural cases it is possible to support different space–time transformations.

5. Results

5.1. Experimental setup

The proposed HLS approach has been tested generating processor arrays for three different algorithms: MatMul algorithm, and Cholesky and LU decomposition algorithms. The MatMul algorithm is high external data demanding, whereas the Cholesky and LU algorithms have non-rectangular loop bounds. The space–time mapping for MatMul was derived by using the scheduler function $\vec{\lambda} = [1\ 1\ 1]$, and the projection vector $\vec{u} = [1\ 0\ 0]^t$; the mapping for Cholesky was obtained from the scheduler $\vec{\lambda} = [1\ 1\ 1]$, and the projection $\vec{u} = [0\ 0\ 1]^t$; and the space–time mapping for LU was obtained from $\vec{\lambda} = [1\ 1\ 1]$, and $\vec{u} = [0\ 0\ 1]^t$. The iteration interval for these arrays is equal to the most time expensive operation presented in the algorithm. In the case of MatMul, multiplication and addition require one clock cycle, thus one iteration interval is equal to one clock cycle, whereas in the Cholesky and LU cases, the iteration interval is equal to 21 clock cycles due to the division latency. Each one of these implementations has a data and control word of 32-bit and 11-bit respectively. According to [31], with a control word of 11-bit, problem sizes of $N \times N$, where $1 < N < 372$, are possible to be solved. Finally, the memory banks required in the memory system are assumed to be off-chip memories.

The MatMul, Cholesky and LU processor arrays have been placed and routed with Xilinx ISE 13.1, and targeted for a Spartan-6 XC6SLXCSG324C FPGA device included in the Digilent Atlys Development Board. This board includes a 128 MByte DDR2 memory with a 16-bit data bus, which was used for storing the input and output matrixes. Also, for comparison purpose, aMicroBlaze soft-processor has been used implementing the same algorithms but in a sequential fashion. The MicroBlaze implementation includes a 64-KB of local memory without cache, and the AXI Bus for peripheral interconnections. Mainly, the soft-processor was used for measuring the loop-kernel execution time including the external memory accesses. The optimization compiling flag was placed in -O3 in order to obtain the maximum compilation effort.

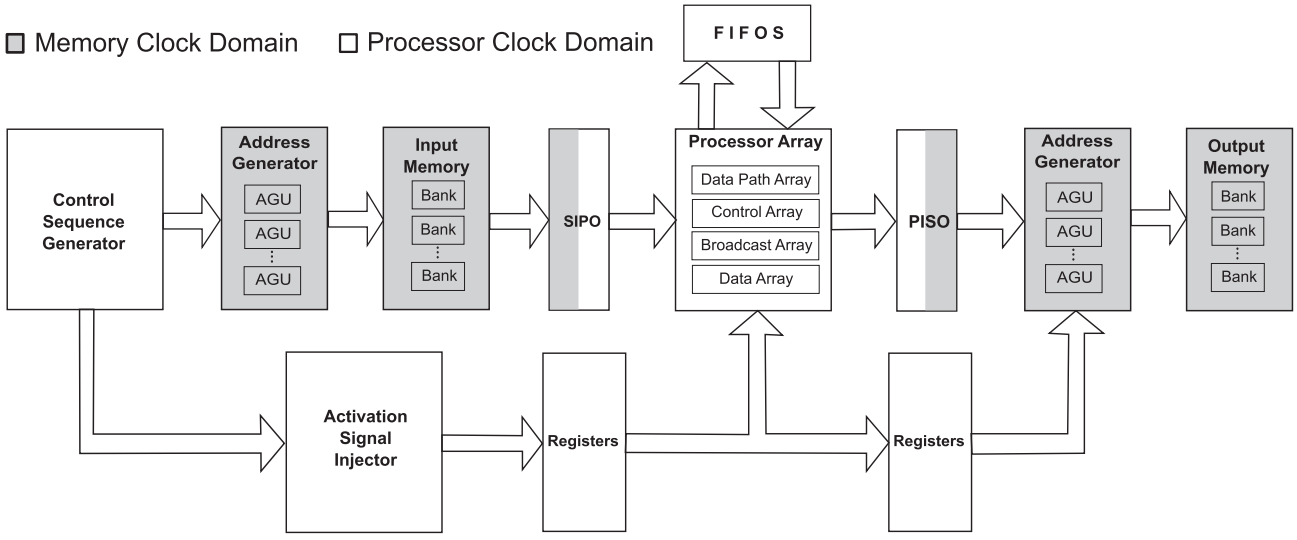

Fig. 8. Complete processor array block diagram, integrating the control, data-path, internal memories (FIFO), and external input/output memories.

Table 2
PAR results for a MicroBlaze implementation and four processor arrays targeted for a XC6SLX45 FPGA device.

FPGA resources		MatMul	MatMul	Cholesky	LU	MicroBlaze processor
Name	Available	2 × 2 PEs	4 × 4 PEs	2 × 2 PEs	2 × 2 PEs	
Slice Regs	54,576	1702	5770	3207	3345	3703
Slice LUTs	27,288	2312	7607	8523	7993	3782
Block RAM	116	116	116	60	60	42
DSP48E1	58	16	42	26	24	3
Max freq (MHz)		45.68	44.62	52.45	51.72	97.75
Power (W)		0.398	0.756	0.334	0.216	0.973

Table 3
Average speed-up and energy consumption per LUT of the four processor arrays.

Metric name	MatMul 2 × 2 PEs	MatMul 4 × 4 PEs	Cholesky 2 × 2 PEs	LU 2 × 2 PEs	MicroBlaze processor
Avg. speed-up	6.05	10.2	5.34	5.18	1
Avg. improvement	24.2	6.5	6.9	11.04	1
mW/LUT	0.172	0.099	0.039	0.027	0.257

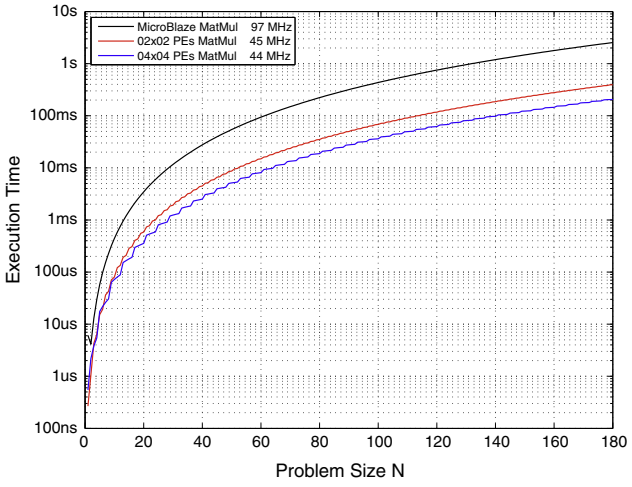

Fig. 9. Comparison of MatMul execution times for two processor arrays and the MicroBlaze implementation.

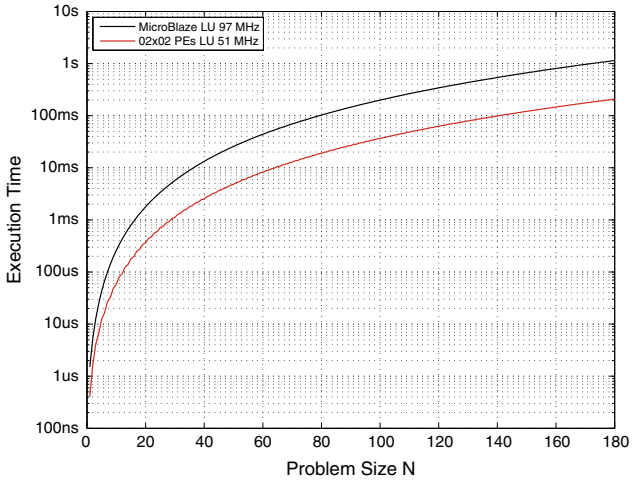

Fig. 11. Comparison of LU execution times for a processor array and the MicroBlaze implementation.

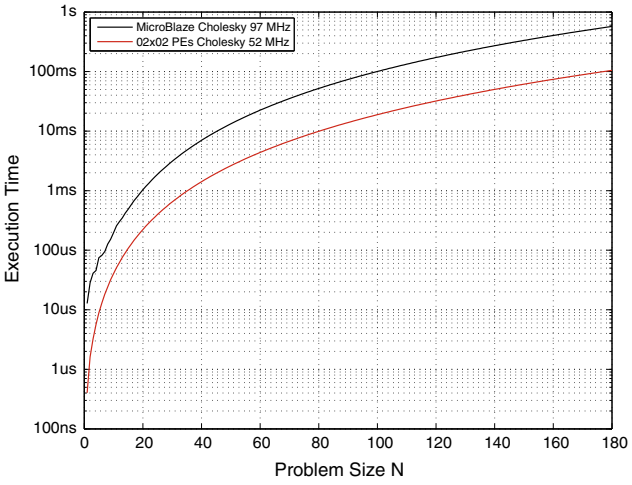

Fig. 10. Comparison of Cholesky execution times for a processor array and the MicroBlaze implementation.

5.2. Implementation results

Table 2 summarizes the place and route (PAR) results for three processor arrays and the MicroBlaze implementation. The FIFO elements required by the processor array are implemented using BRAMs, and the square root and division operations required by the Cholesky and LU decomposition are implemented using the Xilinx's IP cores. Also, Table 2 shows the operational frequency and the dynamic power consumption estimated by the Xilinx's XPower Analyzer using the maximum operational frequencies obtained after PAR, setting the FPGA supply parameters as VCCINT = 1.2, and VCCAUX = 2.5 Volts. Note that the operational frequency of the 4 × 4 array decreases 1% compared against the 2 × 2 array implementation despite the amount of PEs has been quadrupled. Although theoretically with an 11-bit control word the processor arrays are able to solve problem sizes no larger than 342 × 342, there is a memory limitation according to the target device characteristics. In the case of the selected XC6SLX45 FPGA device, the number of BRAMs does not allow to solve problem size bigger than 250 × 250 for the MatMul processor array case, and 180 × 180 for the Cholesky and LU arrays. In the MatMul case, all

the BRAMs are used for storing intermediate data in FIFO memories. On the other hand, since the IP cores used in the Cholesky and LU arrays require BRAMs for implementing the division functionality not all BRAMs could be used for storing data. Moreover, the four processor array implementations consume least power compared to the MicroBlaze processor, since their power operational frequencies are almost 2× slower than in the soft-processor. However, such operational frequency disadvantage is overcome with the fact that the processor arrays have at least 4× more processing elements than the MicroBlaze implementation, and that the processor arrays do not stall their computations in order to access to the external memory; thus a speed-up compared to the soft-processor is achieved.

Table 3 shows the energy consumed by each LUT according to the PAR results shown in Table 2, the average speed-up and the average improvement compared against the MicroBlaze. The improvement is calculated by multiplying the speed-up, the usage of LUTs and the power consumed of each array compared against the MicroBlaze soft-processor. In order to calculate the speed-up some considerations were made. Recall that the memory system tries to provide as many communication channels as the processor array requires. In the case of the MatMul and LU processor arrays three and six 32-bit communication channels are required for the 2×2 and 4×4 PEs respectively. In contrast, two 32-bit communication channels are required for the 2×2 Cholesky processor array. Since the MicroBlaze experimental platform has only one 16-bit communication channel, the speed-up results assume the use of the same one-half communication channel. In this sense, although the operational frequency of the processor arrays is almost 2× slower than the MicroBlaze frequency, an acceleration for the four arrays is achieved. Mainly this is a consequence of that the MicroBlaze processor dedicates more time for performing external memory accesses than the processor arrays. Recall that the processor arrays include a memory system which is in charge of the external memory accesses while the processor array is working, thus the memory system does not stall the processor array computations.

Besides, note that the processor array implementations consume fewer power per LUT compared against the MicroBlaze processor; therefore the processor arrays perform their operations in a more power-efficient way than the soft-processor. With the power per LUT metric (mW/LUT), a more realistic measurement of the power required for performing computation and comparison against a sequential processor implemented in the same technology

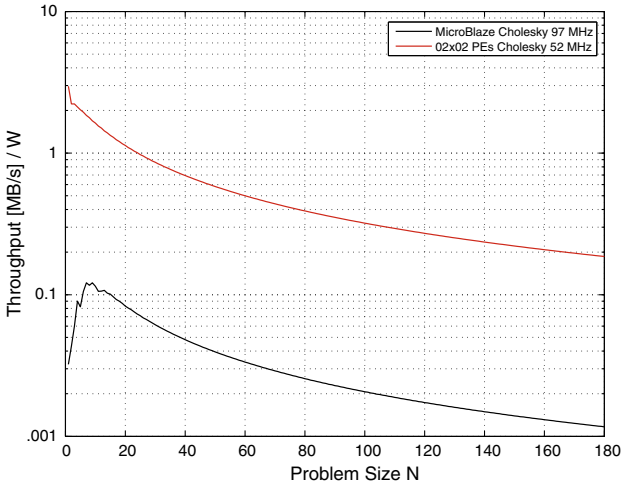

Fig. 13. Cholesky throughput per power unit for a processor array and the MicroBlaze implementation.

can be achieved. In addition, if the speed-up, and usage of LUTs and the total power consumed by each implementation is considered a minimum improvement of 6.5× could be achieved.

Figs. 9–11 show the time required for solving different problem sizes for the MatMul, Cholesky and LU algorithms implemented in processor arrays compared against a MicroBlaze implementation. In these graphics, the *y*-axis represents the execution time in logarithmic scale, while the *x*-axis represents the problem size *N* from 1×1 to 180×180. Note that the execution time achieved by each implementation is less than the time required for their corresponding sequential implementation (MicroBlaze), despite that the MatMul, Cholesky and LU processor arrays have slower clock frequencies. For instance, in the case of the MatMul processor array the execution time achieved for the processor arrays is minor than the time required for their corresponding sequential implementations. Note that the execution time achieved by each implementation is less than the time required for their corresponding sequential implementation, despite that the processor arrays have slower clock frequencies. Specially, in the case of implementation 4×4 MatMul processor array an order of magnitude difference with respect of its sequential implementation is achieved when

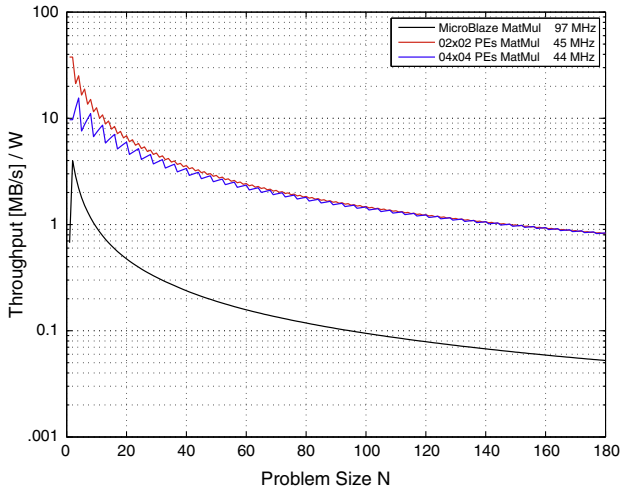

Fig. 12. MatMul throughput per power unit for two processor arrays and the MicroBlaze implementation.

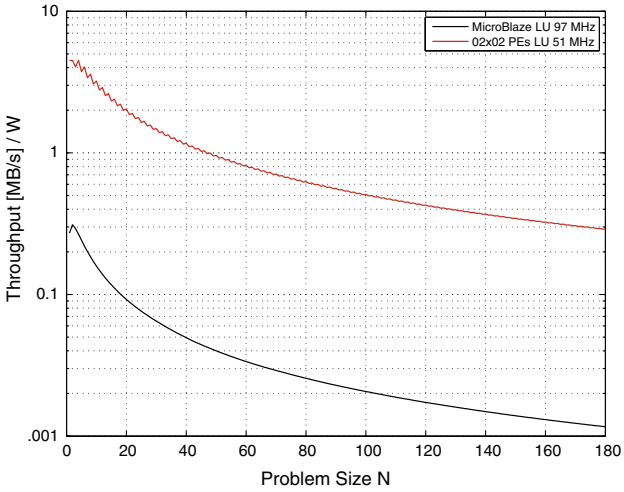

Fig. 14. LU throughput per power unit for a processor array and the MicroBlaze implementation.

$N = 80$. In the cases of Cholesky and LU implementations, when $N = 180$ a speed-up of 5x is achieved.

With the purpose of providing an idea of the execution times compared to a personal computer, a quick comparison against the sequential implementations coded in C, and targeted for a computer with an Intel Xeon at 2.4 GHz, 12 GB in RAM (DDR3) was performed. In summary, the execution time comparison shows an acceleration of 1.1× and 4.2× for the 2×2 and 4×4 MatMul processor arrays, respectively; whereas for the 2×2 Cholesky and LU arrays no execution time improvement is achieved.

Finally, Figs. 12–14 show the throughput per power unit achieved by the processor arrays implementations and their corresponding MicroBlaze implementation. The y-axis represents the [MB/s]/W in logarithmic scale, while the x-axis represents the problem size N from 1×1 to 180×180. In these figures, note that the throughput achieved by the four arrays is similar since the amount of bytes delivered per second is limited by the 16-bit communication channel available in the Atlys Board. Despite this limitation, when $N = 180$ the 4×4 MatMul processor array implementation has an improvement of 2.2× on the throughput per power unit compared against the other two processor array implementations. However, it should be noted that the 4×4 MatMul implementation has four times more PEs than the 2×2 MatMul, Cholesky and LU arrays (due to the 4×4 array has more communication channels). Nevertheless, the throughputs achieved by the MatMul, Cholesky and LU processor arrays are larger than the throughput of their corresponding sequential implementation. In fact, in the Cholesky and LU cases, the throughput achieved when $N = 180$ is one order of magnitude greater than the MicroBlaze implementations.

6. Conclusions

In this paper a HLS approach for generating embedded processor arrays based on the polytope model has been presented. Due to mathematical expressions obtained after space–time transformation are mapped to combinational logic, the proposed approach is able to support different schedulers, allocators and iteration intervals. Moreover, the derived processor arrays are able to solve a set of problem size without the need of generating several arrays for each problem size.

The proposed approach has been validated by generating three different processor arrays for three different cases of study. Unlike previous works, the processor arrays derived by the proposed solution provide support for solving a set of several problem size depending on the memory available in the FPGA, and on the control word width. These processor arrays could be used as generic co-processors for embedded environments due to they are independent of the problem size to be solved. Also, the proposed approach provides a solution for efficient data memory accesses and data transferring for feeding the processor array. Implementation results show that the generated processor arrays achieve an acceleration with respect of a MicroBlaze processor despite their low operational frequency. These low frequencies allow the processor arrays consume fewer power compared against the MicroBlaze processor. Therefore, the processor arrays use efficiently the FPGA computational resources (LUTs), providing an acceleration with a few power consumption compared against the MicroBlaze implementation.

Acknowledgments

First author thanks the National Council for Science and Technology from Mexico (CONACyT) for financial support through the scholarship 3792, and to Dr. Manuel E. Guzman and Dr. Jose Juan Garcia for their discussions and comments about this research.

References

[1] Mojtaba Mehrara, Thomas Jablin, Dan Upton, David August, Kim Hazelwood, Scott Mahlke, Multicore compilation strategies and challenges, IEEE Signal Process. Mag. 26 (6) (2009) 55–63.
[2] Dinesh C. Suresh, Satya R. Mohanty, Walid A. Najjar, Laxmi N. Bhuyan, Frank Vahid, Loop level analysis of security and network applications, in: Proceedings of the 6th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), Anaheim, CA, USA, February, 2003, pp. 44–50.
[3] John G. Proakis, Dimitris K. Manolakis, Digital Signal Processing: Principles, Alogrithms and Applications, fourth ed., Prentice-Hall, 2006.
[4] Peng Liu, Pengeheng Zhu Yan Du, Wei Zhang, A new efficient MIMO detection algorithm based on Cholesky decomposition, in: Proceedings of the 6th International Conference on Advanced Communication Technology (ICACT), Phoenix Park, Korea, February, 2004, pp. 264–268.
[5] Yunhui He, Real-time nonlinear facial feature extraction using Cholesky decomposition and QR decomposition for face recognition, in: Proceedings of the International Conference on Electronic Computer Technology (ICECT), Macau, China, February, 2009, pp. 306–310.
[6] Zille Huma Kamal, Ajay Gupta, Leszek Lilien, Ashfaq Khokhar, An efficient MAP classifier for sensornets, in: Proceedings of the 13th IEEE International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science (LNCS), vol. 4297, Bangalore, India, December 2008, pp 287–293.
[7] Depeng Yang, Gregory D. Peterson, Husheng Li, Junqing Sun, An FPGA implementation for solving least square problem, in: Proceedings of 17th IEEE Symposium on Field Programmable Custom Computing Machines, Napa, CA, USA, April 2009, pp. 303–306.
[8] Hsiao-Chun Wu, Shih Yu Chang, Tho Le-Ngoc, Efficient rank-adaptive least-square estimation and multiple-parameter linear regression using novel dyadically recursive hermitian matrix inversion, in: Proceedings of the International Wireless Communications and Mobile Computing (IWCMC), Crete Island, Greece, August, 2008 pp. 1064–1069.
[9] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, Seth Copen Goldstein, Spatial computation, in: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, MA, USA, October 2004, pp. 14–26.
[10] Rajesh Gupta, Forrest Brewer, High-Level Synthesis: A Retrospective, High-Level Synthesis: From Algorithm to Digital Circuit, Springer Publishing Company, 2008, pp. 13–28 (Chapter 2).
[11] Alex Orailoglu, Daniel D. Gajski, Flow graph representation, in: Proceedings of the 23rd ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, June 1986, pp. 503–509.
[12] Sumit Gupta, Rajesh K. Gupta, Nikil D. Dutt, Alexandru Nicolau, SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits, Kluwer Academic Publishers, 2004.
[13] Shail Aditya, Vinod Kathail, Algorithmic Synthesis Using PICO: An Integrated Framework for Application Engine Synthesis and Verification from High Level C Algorithms, High-Level Synthesis: From Algorithm to Digital Circuit, Springer Publishing Company, 2008, pp. 53–74 (Chapter 4).
[14] Christian Lengauer, Loop parallelization in the polytope model, in: Proceedings of the 4th International Conference on Concurrency Theory (CONCUR), Lecture Notes in Computer Science (LNCS), vol. 715, Hildesheim, Germany, August 2008, pp. 398–416.
[15] Harald Devos, Kristof Beyls, Mark Christiaens, Jan V. Campenhout, Erik H. D'Hollander, Dirk Stroobandt, Finding and applying loop transformations for generating optimized FPGA implementations, in: Transactions on High-Performance Embedded Architectures and Compilers I, Lecture Notes in Computer Science (LNCS), vol. 1, 2007, pp. 159–178.
[16] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, Jürgen Teich, PARO: synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications, in: Proceedings of the 4th International Workshop on Applied Reconfigurable Computing (ARC), Lecture Notes in Computer Science (LNCS), vol. 4943, London, UK, March 2008, pp. 287–293.
[17] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, Tanguy Risset, High-level synthesis of loops using the polyhedral model, in: The MMAlpha Software, High-Level Synthesis: From Algorithm to Digital Circuit, Springer Publishing Company, 2008, pp. 215–230 (Chapter 12).
[18] Alexandru Plesco, Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators, PhD Thesis, École Normale Supérieure de Lyon, September 2010.
[19] Steven Derrien, Platforms, Methodologies and Tools for Designing Reconfigurable Hardware Architectures, PhD Thesis, L'Université de Rennes 1, December 2011.
[20] Frank Hannig, Scheduling Techniques for High-Throughput Loop Accelerators, PhD Thesis, University of Erlangen-Nuremberg, Germany, August 2009.
[21] Hritam Dutta, J. Zhai, Frank Hannig, Jürgen Teich, Impact of loop tiling on the controller logic of acceleration engines, in: Proceedings of the 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors, Boston, MA, USA, July 2009, pp. 161–168.

[22] Hritam Dutta, Mapping of Hierarchically Partitioned Regular Algorithms onto Processor Arrays, Master Thesis, University of Erlangen-Nuremberg, October 2004.
[23] Holger Ruckdeschel, Hritam Dutta, Frank Hannig, Jürgen Teich, Automatic FIR filter generation for FPGAs, in: Proceedings of the 5th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), Samos, Greece, July 2005, pp. 51–61.
[24] Hritam Dutta, Synthesis and Exploration of Loop Accelerators for Systems-on-a-Chip, PhD Thesis, University of Erlangen-Nuremberg, Germany, 2011.
[25] Uday Bondhugula, J. Ramanujam, Ponnuswamy Sadayappan, Automatic mapping of nested loops to FPGAs, in: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), San Jose, CA, USA, March 2007, pp. 101–111.
[26] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, Mukund Sivaraman, Pico-NPA: high-level synthesis of nonprogrammable hardware accelerators, J. VLSI Sig. Proc. 31 (2) (2002) 127–142.
[27] Pico technology. <http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/SynphonyCCompiler.aspx> (Last seen in May, 2014).
[28] Christophe Alias, Bogdan Pasca, Alexandru Plesco, FPGA-specific synthesis of loop-nests with pipelined computational cores, Microprocess. Microsyst. 36 (8) (2012) 606–619.
[29] Florent de Dinechin, Bogdan Pasca, Designing custom arithmetic data paths with FloPoCo, IEEE Des. Test Comput. 28 (4) (2011) 18–27.
[30] Alain Darte, Leonid Khachiyan, Yves Robert, Linear scheduling is nearly optimal, Parallel Process. Lett. 1 (2) (1991) 73–81.
[31] Roberto Perez-Andrade, Cesar Torres-Huitzil, Rene Cumplido, Juan M. Campos, On a hybrid and general control scheme for algorithms represented as a polytope, in: Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPSW), Anchorage, AK, USA, March 2011, pp. 330–333.
[32] Roberto Perez-Andrade, Cesar Torres-Huitzil, Rene Cumplido, Juan M. Campos, On an external memory scheme for processor arrays, IEICE Electron. Express 10 (14) (2013) 20130324.

Roberto Perez holds the BSc degree in Computer Engineering and the MSc in Computer Science. In 2014 he received the PhD degree in Computer Science from Center of Research and Advanced Studies of the National Polytechnic Institute (CINVESTAV), of Mexico. His areas of interest includes the design of hardware architectures and embedded systems using configurable computing platforms for their validation as well as high-level synthesis techniques involved during the hardware architecture design process.

Cesar Torres is a full time researcher in computer science at the Information Technology Laboratory at the Center of Research and Advanced Studies of the National Polytechnic Institute (CINVESTAV), of Mexico. He holds the BSc degree in Electronics and received the MSc degree in Electronics. In 2003 he received the PhD degree in Computer Science from the National Institute for Astrophysics, Optics, and Electronics of Mexico. His research interests include computational vision, reconfigurable computing, FPGAs and their computational applications, bio-inspired systems, and embedded computer systems.

Rene Cumplido holds a PhD in electrical engineering from Loughborough University, UK (2001). Since 2002, he is member of the Reconfigurable and High Performance Computing Group at INAOE, Mexico. His research interests are Reconfigurable Computing, FPGA Technologies, and Custom Architectures. He has published more than 90 scientific papers in international conferences and journals. He is co-founder and general chair of the International Conference on Reconfigurable Computing and FPGAs, ReConFig. He is the founder editor-in-chief of the International Journal of Reconfigurable Computing, and associate editor of several international journals on the fields of computer and electrical engineering.