Processor Arrays Generation for Matrix Algorithms Used in Embedded Platforms

Roberto Perez-Andrade, Cesar Torres-Huitzil
Information Technology Laboratory
Advanced Studies Center of the National
Polytechic Institute, CINVESTAV
Ciudad Victoria, Mexico
email: {jrperez, ctorres}@tamps.cinvestav.mx

Rene Cumplido, Juan M. Campos
Department of Computer Science
National Institute for Astrophysics,
Optics and Electronics, INAOE
Santa Maria Tonantzintla, Puebla, Mexico
email: {rcumplido, jcampos}@ccc.inaoep.mx

***Abstract*—Matrix algorithms are an important part of many digital signal processing applications as they are core kernels that usually need to be applied many times. Hardware assisted implementations using FPGAs provide a good compromise between performance, cost and power consumption. This paper presents a high level synthesis approach to generate embedded processor arrays for matrix algorithms based on the polytope model. The proposed approach provides a solution for efficient data memory accesses and data transferring for feeding the processor array, as well as it provides support for solving a set of problem size depending on FPGA available resources. The proposed approach has been validated by generating processor arrays for two case studies targeted for a Spartan-6 device. Results show that the implemented array outperforms hardware and software implementations targeted to embedded platforms as well.**

I. Introduction

In an embedded computing scenario, a balance between power consumption and computational performance is desired. For instance, a mobile phone supporting video playback has a strict power budget, but it also has to meet certain performance requirements [1]. Traditionally, such computational performance were met by technology advances, until problems like power dissipation and thermal constraints emerged as dominant design issues and forced computer designers away from relying on increasing frequency to improve performance with low power consumption. Using multiple processing units for performing parallel computations and completing a larger volume of work in shorter time periods has been a current trend in order to improve performance [2]. Computer designers can rely on exploiting different forms of parallelism such as instruction level parallelism, data level parallelism or loop level parallelism (LLP) in order to achieve the performance required by an application.

Loop level parallelism approach is a popular parallelization method used in digital signal processing because several electronic systems used in this domain are built on loop based algorithms like matrix multiplications, matrix decompositions, convolutions, and system of equations solvers [3, 4, 5]. Designers are often directly responsible for crafting the application-specific computing architecture, the control and the modules that manage, and transport data to/from the processing kernels leading to spend a considerable time and effort during the design. Hence, a hardware assisted approach to automatically generate dedicated hardware architectures might be beneficial in several applications. High-level synthesis (HLS) methods allow a fast exploration and evaluation of possible hardware architectures. Generally, HLS methods try to extract automatically the parallelism from an algorithmic representation, and at the same time, they derive parallel hardware structures from this algorithmic input. One of the representations used for HLS is the polytope model [6], which provides 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 processor arrays [7, 8], or highly-pipelined mono-processors [9].

In this paper, a HLS approach for generating embedded 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 II presents a general overview of the related works concerning the processor array generation within the polytope model. A brief description of the processor array generation using the polytope model is presented in section III. Section IV 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 and acceleration results compared against a soft-processor implementation for two algorithms are presented in section V, while the conclusions are presented in section VI.

II. Related Work

One of the works about HLS within the polytope scope is the MMAlpha programming environment [8], which transforms an input specification in form of a system of uniform recurrence equations (SURE) to VHDL code describing a processor array. 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). In [10], Plesco presents a hand-made solution for interfacing an external

978-1-4799-2079-2/13/$31.00

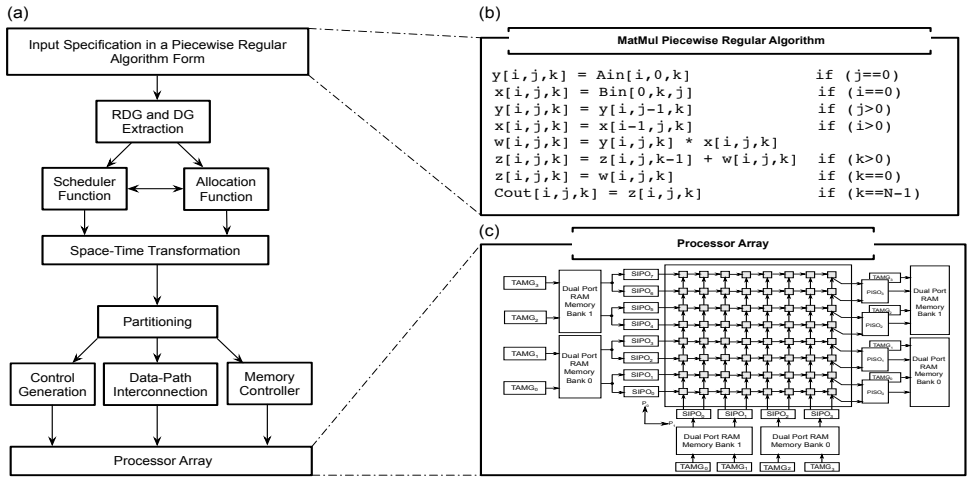

Fig. 1. (a) Processor Array Design Flow Followed in this Research. (b) Matrix-Matrix Multiplication PRA. (c) Processor Array Derived by the Design Flow.

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 [11] 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 [7], which is able to map computational intensive nested loop programs into parallel architectures that are translated into VHDL code. 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 [12]. Moreover, PARO cases of study include MatMul, FIR filter, discrete cosine transform and images filters.

Uday *et al.* in [13] present a framework for mapping perfectly nested loops into processor arrays implemented in FPGAs. In their 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. 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.

Although tools like PARO and MMAlpha produce processor arrays with different allocation methods (like projection or partitioning) they lack of parametric support, *i.e.* if the problem size for which they were targeted changes, it is needed to regenerate the processor array. 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 [10]. In the following section, the design flow using the polytope model is described.

III. 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 $S_i[I]$ is defined for all $I \in \mathcal{I}_i$ according to:

$$x_i\left[P_i\vec{I}\right] = \mathcal{F}_i\left(..., x_j\left[Q_j\vec{I} - \vec{d}_{ji}\right], ...\right) \quad if \quad \mathcal{C}_i^I\left(\vec{I}\right)$$

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

The common design flow followed for the generation of processor arrays within the polytope model is shown in Figure 1.a. First, the original program, or *source polytope* is represented as a PRA (Figure 1.b). From the source polytope, a reduced dependence graph (RDG) and the iteration space $\mathcal{I}$ are extracted so as to define a scheduler and an allocation functions. The purpose of the scheduler is to assign a computation

date for each task (*i.e.* $\vec{I}$), whereas the allocation assigns the tasks to PEs such as no two tasks with the same computation date are assigned to the same PE. The linear scheduling proposed in [14] is used within the context of this work, since it derives asymptotically functions equivalent to the best scheduler. On the other hand, the allocation function used in this work is obtained from a user proposed projection vector $\vec{u}$. Also, a time interval between two successive computations performed by the same PE, called *iteration interval*, is calculated from the scheduler and allocation 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 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. After the space-time transformation, the space indexes are partitioned in order to obtain a processor array of a fixed size preserving the interconnection topology among PEs. In the proposed approach, strip mining is used to partition each dimension of the iteration 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 SSp_k, and it adds new dimensions for scanning them without adding these indexes to the PRA [15]. The stride size SSp_0 defines the size of a one-dimensional processor array, whereas SSp_0 and SSp_1 define the size of a two-dimensional array, *i.e.* $\mathcal{P} \subset \mathbb{Z}$ and $\mathcal{P} \subset \mathbb{Z}^2$, respectively. After these transformations, the controller [7, 8, 15], the processor array topology [16, 17], the PE data-path [7, 8] and the memory controller [10, 18] are synthesized (Figure 1.c). 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 specify 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. 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. The next section describes the generation of the processor array following the design flow shown in Figure 1.a.

IV. Architectural Design

A. Processing Element Data-Path

The interconnection 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. Besides of the PE interconnection, it is also needed to determinate the amount of delay elements (registers) that should be placed between these interconnections. The amount of registers is derived by using the scheduler function and data dependences. The PEs data-path is derived by binding the RDG obtained from the input algorithmic representation to functional units (like adders, multipliers, etc). Some multiplexers are added in cases that there exist several nodes corresponding to the same indexed variable. More details about the PE synthesis can be found in [16, 17].

B. Controller

The processor array controller is based on a centralized and distributed approach (Figure 2). 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 [15]. However, for sake of completeness, the next subsection briefly summarize the three controller modules.

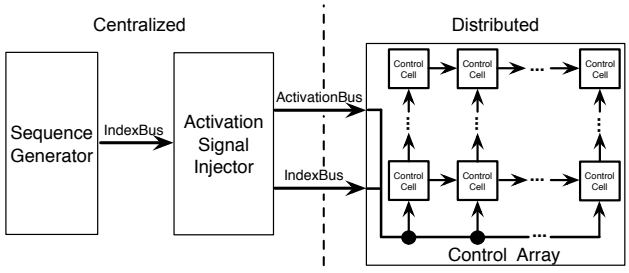

Fig. 2. Processor Array Controller Block Diagram.

1) 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 according to the values calculated by a combinational submodule. The Max/Min submodule calculates the maximum and minimum values of such ranges. 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.

2) 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. It also 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.

3) The Control Array: 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 when a PE maps correctly an iteration point $\vec{I}$ from the processor space $\mathcal{P}$. The second characteristic helps to support algorithms with non-rectangular processor spaces, because when a non-rectangular processor space is partitioned there are some cases where the PEs should remain inactive while a partition is being scanned. The control array is composed by a set of control cells. Each control cell includes combinational logic that checks the correct mapping of the processor space and some counters for generating the activation pattern.

C. Memory System

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 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. A border mapping occurs when the index vector $\vec{I}$ of $\mathcal{C}^I(\vec{I})$ 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 $\mathcal{C}^I(\vec{I})$ 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. Together, there are four possible architectural cases: input border mapping (Figure 3.a), output border mapping (Figure 3.b), input broadcast mapping (Figure 3.c), and output broadcast mapping (Figure 3.d).

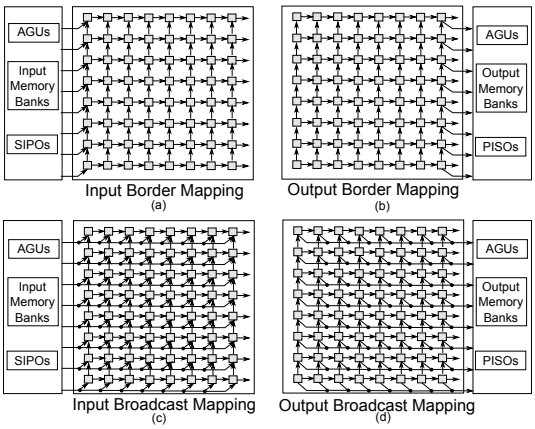

Fig. 3. Architectural Cases According to the Variable Type and the Mapping Possibilities.

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. 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 (Clk_{mem}) with respect to the processor array clock frequency (Clk_{pa}), *i.e.* $Clk_{mem} = 2 \times Clk_{pa}$. 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. Due to the four mapping cases, derived from the I/O variables and its iteration dependent condition, there are also four different architectural modules. Figure 4 exemplifies the internal architecture of two of these four cases. A more detailed explanation of the complete memory system could be found in [18].

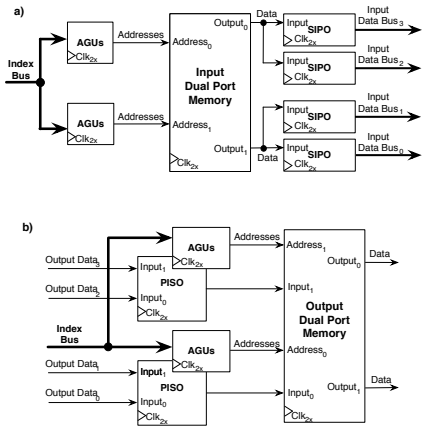

Fig. 4. Interconnection Example of the a) Input Broadcast Case and b) Output Border Case.

D. Integrated System

Figure 5 shows the block diagram integrating the processor array data-path, the controller, and the memory system. 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 and by the AGUs in order to generate the processor array activation sequence and for extracting data from an input memory. Data extracted from memory is 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 cache 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 mention that by changing some mathematical expressions presented in the sequence generator, the activation-signal injector, and the AGUs, it is possible to support different scheduler functions. 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.

V. Results

A. Experimental Setup

The proposed HLS approach has been tested generating processor arrays for two different algorithms: MatMul and Cholesky decomposition. The MatMul algorithm is high external data demanding, whereas the Cholesky algorithm has non-rectangular loop bound. The space-time mapping for MatMul

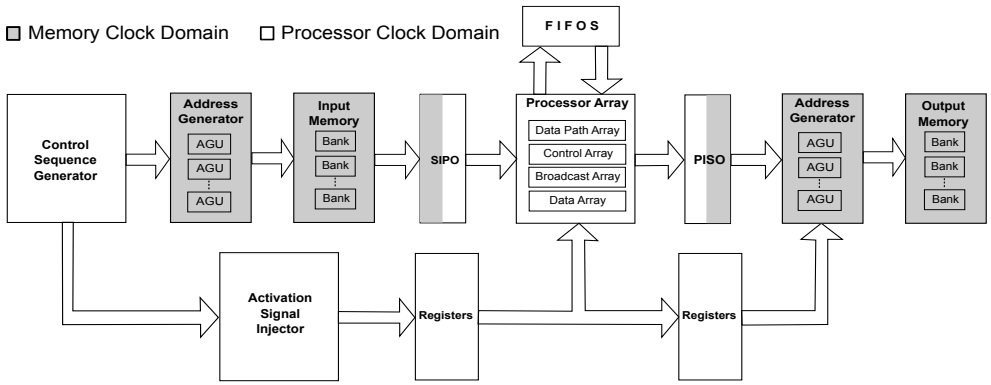

Fig. 5. Complete Processor Array Block Diagram, Integrating the Control, Data-Path, Internal Memories (FIFO), and External Input/Output Memories.

was derived by using the scheduler function $\vec{\lambda} = [1\ 1\ 1]$, and the projection vector $\vec{u} = [1\ 0\ 0]^t$, whereas the mapping for Cholesky was obtained from the scheduler $\vec{\lambda} = [1\ 1\ 1]$, and the projection $\vec{u} = [0\ 0\ 1]^t$. 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 case, 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 [15], 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 and Cholesky 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, a MicroBlaze 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.

B. Implementation Results

Table I 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 decomposition are implemented using the Xilinx's IP cores. Also, table I shows the operational frequency and the power consumption estimated by the Xilinx's XPower Analyzer. 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 bigger than 371×371, there is a memory limitation according to the target device characteristics. The selected FPGA device, the amount 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 array. 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 array require BRAMs for implementing the division functionality not all BRAMs could be used for storing data.

TABLE I. PAR RESULTS FOR A MICROBLAZE IMPLEMENTATION AND THREE PROCESSOR ARRAYS TARGETED FOR A XC6SLX45 FPGA DEVICE.

FPGA Resources Name	Available	MatMul 2×2 PEs	MatMul 4×4 PEs	Cholesky 2×2 PEs	MicroBlaze Processor
Slice Regs	54,576	1,702	5,770	3,207	3,703
Slice LUTs	27,288	2,312	7,607	8,523	3,782
Block RAM	116	116	116	60	42
DSP48E1	58	16	42	26	3
Max. Frequency (MHz)		45.68	44.62	52.45	97.75
Power Consumption (W)		0.398	0.756	0.334	0.973

TABLE II. AVERAGE SPEED-UP AND ENERGY CONSUMPTION PER LUT OF THREE PROCESSOR ARRAYS.

Metric	MatMul 2×2 PEs	MatMul 4×4 PEs	Cholesky 2×2 PEs	MicroBlaze Processor
AVG. Speed-Up	6.05	10.20	5.34	1
mW/LUT	0.172	0.099	0.039	0.257

Table II shows the average speed-up compared against the MicroBlaze implementation and the energy consumed by each LUT according to the PAR results. Recall that the memory system tries to provide as many communication channels as the processor array requires. In the case of the MatMul 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 two times slower than the MicroBlaze frequency, an acceleration for the three arrays is achieved. Moreover, the processor arrays are more power-efficient than the MicroBlaze implementation, since they consume fewer energy per LUT despite they require more computational resources.

Figure 6 shows the time required for solving different problem sizes for the MatMul and Cholesky algorithms implemented in processor arrays compared against their MicroBlaze implementations. 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 for the processor arrays is minor than the time required for their corresponding sequential implementations. Finally, a quick comparison against sequential implementations coded in C, and targeted for a personal computer with an Intel Xeon at 2.4 GHz, 12 GB in RAM (DDR3), shows an acceleration of 1.1x and 4.2x for the 2×2 and 4×4 MatMul processor arrays, respectively; whereas for the 2×2 Cholesky array no execution time improvement is achieved.

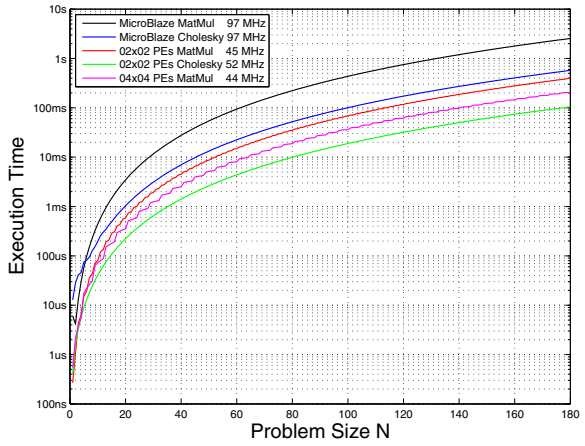

Fig. 6. MatMul and Cholesky Execution Times for the Processor Arrays and their MicroBlaze Implementations.

VI. Conclusion

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 two 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 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. However, 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.

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 for his contribution to this research.

References

[1] G. Blake, R. G. Dreslinski, and T. Mudge, "A Survey of Multicore Processors," *IEEE Signal Processing Magazine*, Vol. 26, No. 6, pp. 27–37, 2009.

[2] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke, "Multicore Compilation Strategies and Challenges," *IEEE Signal Processing Magazine*, Vol. 26, No. 6, pp. 55–63, 2009.

[3] Y. He, "Real-Time Nonlinear Facial Feature Extraction using Cholesky Decomposition and QR Decomposition for Face Recognition," in *International Conference on Electronic Computer Technology*, pp. 306–310, 2009.

[4] H.-C. Wu, S. Y. Chang, and T. Le-Ngoc, "Efficient Rank-Adaptive Least-Square Estimation and Multiple-Parameter Linear Regression using Novel Dyadically Recursive Hermitian Matrix Inversion," in *International Wireless Communications and Mobile Computing Conference, 2008. IWCMC '08*, pp. 1064–1069, 2008.

[5] D. Xia, H. He, Y. Xu, and Y. Cai, "A Novel Construction Scheme with Linear Encoding Complexity for LDPC Codes," in *4th International Conference on Wireless Communications, Networking and Mobile Computing, 2008. WiCOM'08.*, pp. 1–4, 2008.

[6] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, 2008.

[7] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications," in *Proceedings of the Fourth International Workshop on Applied Reconfigurable Comuting*, Vol. 4943. Springer Verlag, pp. 287–293, 2008.

[8] S. Derrien, S. Rajopadhye, P. Quinton, and T. Risset, *High-Level Synthesis of Loops Using the Polyhedral Model The MMAlpha Software*. Springer, ch. 12, pp. 215–230, 2008.

[9] H. Devos, K. Beyls, M. Christiaens, J. V. Campenhout, E. H. D'Hollander, and D. Stroobandt, "Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations," *Transactions on High-Performance Embedded Architectures and Compilers I*, Vol. 1, pp. 159–178, 2007.

[10] A. Plesco, "Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators," Ph.D. dissertation, École Normale Supérieure de Lyon, 2010.

[11] S. Derrien, "Platforms, Methodologies and Tools for Designing Reconfigurable Hardware Architectures," Ph.D. dissertation, L'Université de Rennes 1, 2011.

[12] H. Dutta, J. Zhai, F. Hannig, and J. Teich, "Impact of Loop Tiling on the Controller Logic of Acceleration Engines," *Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, pp. 161–168, 2009.

[13] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "Automatic Mapping of Nested Loops to FPGAs," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 101–111, 2007.

[14] A. Darte, L. Khachiyan, and Y. Robert, "Linear Scheduling Is Nearly Optimal," *Parallel Processing Letters*, Vol. 01, No. 02, pp. 73–81, 1991.

[15] R. Perez-Andrade, C. Torres-Huitzil, R. Cumplido, and J. M. Campos, "On an Hybrid and General Scheme for Algorithms Represented as a Polytope," in *IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pp. 330–333, 2011.

[16] F. Hannig, "Scheduling Techniques for High-Throughput Loop Accelerators," Ph.D. dissertation, University of Erlangen-Nuremberg, Germany, 2009.

[17] H. Dutta, "Synthesis and Exploration of Loop Accelerators for Systems-on-a-Chip," Ph.D. dissertation, University of Erlangen-Nuremberg, Germany, 2011.

[18] R. Perez-Andrade, C. Torres-Huitzil, R. Cumplido, and J. M. Campos, "On an External Memory Scheme for Processor Arrays," *IEICE Electronics Express*, Vol. 10, No. 14, pp. 20130324–20130324, 2013.