On an external memory scheme for processor arrays

Roberto Perez-Andrade[1a)], **Cesar Torres-Huitzil**[1], **Rene Cumplido**[2], **Juan M. Campos**[2]

[1] *CINVESTAV IPN, Information Technology Laboratory, Tamps., México*

[2] *INAOE, Computer Science Department, Puebla, México*

a) *jrperez@tamps.cinvestav.mx*

Abstract: The problem of generating memory interfaces between loop-based accelerators and external memory is gaining the attention from the high-level synthesis research community. This paper presents an external memory system for inserting/extracting data to/from a loop-based accelerator derived by a high-level synthesis approach. The memory system is composed by four architectural cases which could occur during hardware synthesis. The memory system is based on a global asynchronous local synchronous approach and the use of dual-port memory banks. FPGA-based implementation results show that the proposed memory system is technologically achievable and provides a high-bandwidth without introducing communication overhead.

Keywords: Processor arrays, loop-based algorithms, external memory interface, FPGA

Classification: Electronic devices, circuits, and systems

References

[1] P. Coussy and A. Morawiec, Book *High-Level synthesis from algorithm to digital circuits*, Springer Publishing Company, 2008.

[2] F. Hannig, H. Ruckdeschel, H. Dutta, and J. Teich, "PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications". *Proc. 4th Int. Workshop on Applied Reconfigurable Comuting*, London, UK, volume 4943, pp. 287–293, March 2008.

[3] S. Derrien, S. Rajopadhye, P. Quinton, and T. Risset, "High-Level synthesis of loops using the polyhedral model: the MMAlpha software" in *High-Level synthesis from algorithm to digital circuits*, by P. Coussy and A. Morawiec, pp.215–230, Springer, 2008.

[4] 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". *Trans. High Performance Embedded Architectures and Compilers I*, vol. 4050, pp. 159-178, 2007.

[5] H. Dutta, F. Hannig and J. Teich, "Controller synthesis for mapping partitioned programs on array architectures". *Proc. 19th Int. Conf. Architecture of Computing Systems*, Frankfurt am Main, Germany, pp. 176–191, March, 2006.

[6] A. C. Guillou, P. Quinton, T. Risset, "Hardware synthesis for multi-dimensional time", *Proc. 14th IEEE Int. Conf. Application-Specific Systems, Architectures and Processors*, The Hague, Netherlands, pp. 40–50, Jun, 2003.

DOI: 10.1587/elex.10.20130324
Received May 04, 2013
Accepted June 14, 2013
Publicized June 28, 2013

[7] R. Perez-Andrade, C. Torres-Huitzil, R. Cumplido and J.M. Campos, "On a hybrid and general control scheme for algorithms represented as a polytope". *Proc. 25th IEEE Int. Symp. Parallel and Distributed Processing Workshops and Phd Forum*, Anchorage, USA, pp. 330–333, May, 2011.
[8] S. Derrien, "Automation for Applictaion Specific Hardware Architectures". *Research Habilitation*, Université de Rennes 1, France, 2011.
[9] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, G. Snider "High-Level Synthesis of Nonprogrammable Hardware Accelerators". *Proc. 11th IEEE Int. Conf. Application-Specific Systems, Architectures and Processors*, Boston, USA, pp.113–124, Jul, 2000.
[10] H. Dutta, J. Zhai, F. Hannig and J. Teich, "Impact of loop tiling on the controller logic of acceleration engines". *Proc. 20th IEEE Int. Conf. Application-Specific Systems, Architectures and Processors*, Boston, USA, pp. 161–168, Jul, 2009.
[11] A. Plesco, "Program transformations and memory architecture optimizations for high-level synthesis of hardware accelerators". *PhD thesis*, École Normale Supérieure de Lyon, France, 2010.

1 Introduction

Loop level parallelism (LLP) 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. Implementing highly-parallel hardware architectures exploiting the LLP by hand is cumbersome, error-prone, and it leads to spend much time during the design space exploration phase. 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 the input representation. One of the representations used for HLS [1] is the polytope model, 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 [2, 3], or highly-pipelined mono-processors [4].

Processor arrays are regular, locally connected and massively parallel architectures with simple processing elements (PEs), whose structure is well-suited for their implementation in VLSI or FPGAs. The synthesis of the processor array interconnection topology, the PE data-path [2], and the derivation of control structures are topics well studied [5, 6, 7]. However, there are few attempts for deriving memory interfaces for processor arrays constructed by using the polytope model. In this paper, an external memory interface system for inserting/extracting data to/from the processor array based in four architectural cases is proposed. In the rest of this paper, we will refer as external memory as the memory placed outside of the processor array, mainly to store input and output data. Such external memory interface sys-

tem could be implemented by using built-in FPGA Block-RAMs (BRAMs) or by off-chip DRAM. Our proposed memory scheme is based on the assumption that all external data are required and produced by the processor array during each clock cycle respecting the algorithm data dependences. This assumption can be interpreted as the worst case scenario when the processor array is derived, and it guarantees that highly data demand algorithms could be supported by the memory scheme without adding latency. The impact of such assumption in the memory scheme is that the external data are distributed into different memory banks working in parallel, and in a different clock domain compared to the processor clock. The proposed memory system could be used as a complement of HLS tools within the polytope model context like [2, 3].

2 Related work

The HLS research community has mostly focused on deriving hardware accelerators from high-level specifications. The problem of automatically generating memory interfaces for processor arrays has received little attention [8]. Consequently, the derivation of memory interfaces between processor arrays and external memory devices is gaining attention from the research community. First attempts for solving the data I/O were based on ad-hoc arbitrator mechanisms implemented in hardware and controlled by a host [9].

Dutta *et al.* in [10] provide a methodology for automatic generation of control engines in charge of orchestrating data transfer and computations for processor arrays generated by the PARO framework [2]. They describe a scheme for the memory controller synthesis based on the use of counters, decoders, address generators, and glue logic for interfacing the processor array to other components integrated in a system-on-a-chip (SoC) environment. 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. In [11], Plesco presents a hand-made solution for interfacing an external memory with a processor array of 4×4 PEs generated by MMAlpha tool [3] by using the matrix-matrix multiplication (MatMul) of complex numbers of 32-bit word size. This hand-made solution is only an specific implementation without any generalization for other cases of study. Plesco states that besides the difficulty of designing a memory architecture, knowledge of the processor array data access patterns, and enable an internal data reuse are needed in order to obtain a good performance. Also, within the MMAlpha framework and using the MatMul algorithm, Derrien in [8] 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. A common factor in these related works is that, latency penalties or implement I/O serialization for external data are introduced, contrary to our proposed memory system that provides data as it is needed, without stalling the processor array execution.

3 Polytope model overview

The polytope model provides an abstraction to represent computations in a sequential loop program or in a more general representation called system of uniform recurrence equation (SURE). The SURE concept has originated several cases of recurrence equations like the piecewise regular algorithm (PRA) since it is more specific than a SURE and it describes the case when conditional statements inside of a loop nest are presented. 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:

$$x_i\left[P_i\vec{I}\right] = \mathcal{F}_i\left(...,x_j\left[Q_j\vec{I}-\vec{d}_{ji}\right],...\right) \quad if \quad \mathcal{C}_i^I\left(\vec{I}\right) \tag{1}$$

where x_i, x_j are affinely indexed variables. The indexing functions of the variables are defined by the constant indexing identity matrices 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 Fig. 1.a. First, the original program, or *source polytope* is represented as a PRA (Fig. 1.b). From the source polytope, a reduced dependence graph 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. 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. Strip mining is used to partition one dimension of the iteration space into strips. 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 [7]. 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. Finally, the controller, the processor array topology, and the PE data-path [2, 3] and the memory controller (Fig. 1.c) are synthesized. In the next section, the proposed solution for the external memory scheme is presented by using the I/O variables presented in the PRA, and their $\mathcal{C}_i^I(\vec{I})$. The PRA I/O variables are denoted by the Ain, Bin and $Cout$ variables in Fig. 1.b.

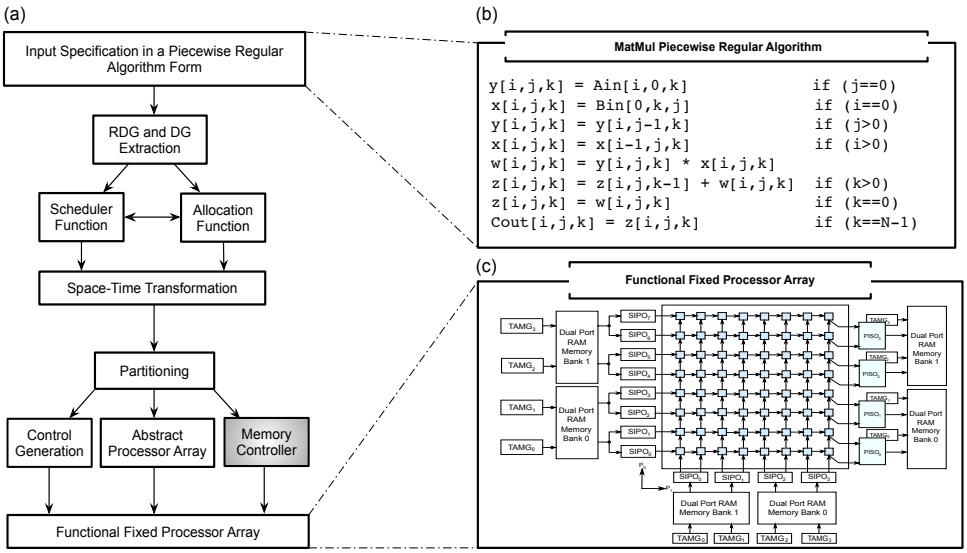

Fig. 1. From the design flow on the polytope model, it is possible to synthesize hardware from a PRA.

4 External memory interface architecture

The derivation of the external memory interface depends on the I/O variables and its iteration dependent condition after space-time. These I/O variables can be viewed as representation of the external memory. Mainly, after space-time mapping, 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 (Fig. 2.a), output border mapping (Fig. 2.b), input broadcast mapping (Fig. 2.c), and output broadcast mapping (Fig. 2.d).

Independently of any of the four architectural cases, the memory scheme could be composed by address generator units (AGUs), memory banks, registers working in serial-input/parallel-output (SIPO) and parallel-input/serial-output (PISO) fashion. The selection of these architectural components, their interconnection, and their internal architecture varies depending on the variable types, the variable index vector $\vec{I}$, the space-time mapping, and the size of the processor array. In the next two subsections the memory banks and the AGU required for all the architectural cases are described

4.1 Memory banks

One of the assumptions made for the external memory interface is that the I/O data is stored in several memory banks organized according to the size of the strips (SSp_k) and the amount of memory banks used. For I/O variables

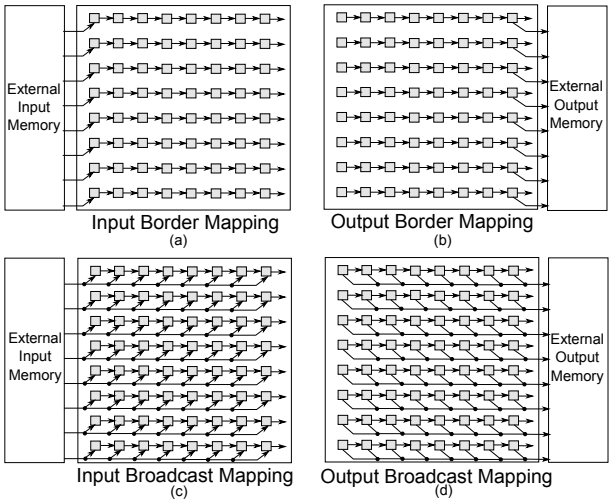

Fig. 2. Architectural cases according to the variable type and the mapping possibilities.

represented as two-dimensional arrays, data are linearized in a row-major or column-major order according to the design constraints. The main idea of the data organization is that a matrix of size $N \times N$ is partitioned into strips of constant size, and at the same time, each strip is divided into $m-1$ data blocks according to the amount of memory banks. The size of these data blocks is given by a memory distribution factor f_x which indicates how many contiguous columns or rows are grouped in a data block. Furthermore, it is assumed the use of dual-port memories for each memory bank and that it is possible to extract two data per memory port in a processor clock cycle. 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.* $Clk_{mem} = 2 \times Clk_{pa}$), similarly to globally-asynchronous locally-synchronous (GALS) approach. Therefore, the combination of both assumptions (GALS approach and dual port memories) leads to have a data extraction rate of four data per memory bank in a processor clock cycle.

4.2 Address generator unit

The function of this module is the generation of the memory addresses for each memory bank given an index bus provided by the controller [7]. An AGU contains a combinational module in charge of generating the addresses following a mathematical formula as shown in Eq. 2.

$$MemAddress_j = (N \times k) + i - [f_{col} \times N\,(tilep_x + MemBank_j)] \qquad (2)$$

where $MemAddress_j$ is the memory address for the *j-th* memory bank, i, and k are the indexes used for a PRA variable, N is the problem size, $MemBank_j$ is the memory bank id, f_{col} is the memory factor distribution and $tilep_x$ is one tile index obtained after partitioning the processor space. Although the AGUs function is the same regardless of the four architectural cases, its internal architecture varies according to the border and broadcast

mapping possibilities. Basically, in both cases the address generator needs to scan its corresponding memory bank according to the scheduler function in order to respect the activation sequence imposed by the space-time mapping. Since in the case of border mapping one of the indexes of the I/O variable is mapped to time space, this index could be used for scanning a memory bank. On the other hand, in the case of the broadcast mapping both variable indexes are mapped to processor space, thus one processor index must be scanned by using counters. Next subsections present one by one the four architectural cases. Each one of these cases are modular in the sense that by replicating each memory interface it is possible to support processor arrays of different size.

4.3 Input variable border mapping

The first architectural case occurs when the space-time mapping indicates that the input variable of the PRA is inserted in a processor array border. After the transformation, the indexes from vector $\vec{I}$ of the input variable are mapped to time space and processor space. However, due to the iteration dependent condition $\mathcal{C}^I(\vec{I})$ one processor space dimension is set to zero whereas the other dimension remains variable. This leads to the idea of placing the AGUs at the border of the constant dimension, and generating the remaining processor index by replicating the same amount of AGUs as the processor array border indicated by SSp_k. As a result of the GALS approach two AGUs per address port are required in order to generate two different addresses in a processor cycle. The selection of the addresses generated by these two AGUs is accomplished by a multiplexer and a control unit. The interconnection of two AGUs, the multiplexer and the flip-flop T is called two-address generator module (TAGM). Since the memory bank produces two data per memory port in a processor clock cycle, one of these data must be stored in one memory clock cycle. Besides, there must be an interface from a faster to a slower domain. Such interfacing is performed by a SIPO interface which receives two data extracted from the memory bank and sends both data in the next processor clock cycle. The SIPO interface consists on two-register pairs controlled by different clock domains. Each pair of registers works on parallel with respect of the other pair. The interconnection of the TAGM, the dual-port memory bank and the SIPO is called input border module (Fig. 3). An input border module is able to provide four data from the memory bank. By replicating this module, it is possible to increase the data rate as needed (larger processor arrays), whereas by relaxing one of the two assumptions (GALS design or dual-port memory), it is possible to decrease the data rate (smaller arrays).

4.4 Output variable border mapping

The second architectural case occurs when the space-time mapping indicates that the output variable of the PRA is extracted from a processor array border. Similarly to the input variable case, after space-time transformation the $\mathcal{C}^I(\vec{I})$ indicates that one processor dimension is set to a constant value

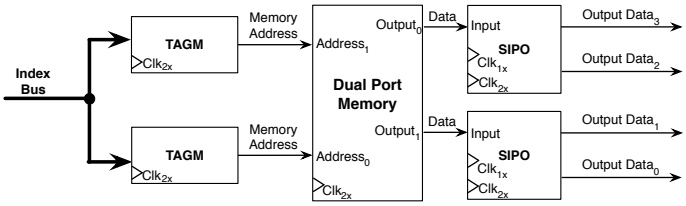

Fig. 3. Interconnection of the TAGM, dual-port memory and SIPO for the input border case.

(problem size) whereas the other dimension remains variable. Therefore, the address generation is tackled in a similar way as the input variable border case, where the constant processor index is generated by replicating the TAGMs. Since each TAGM produces two memory addresses in one processor cycle, it is possible to insert two data produced by the array sharing the same input port. The selection of the two inputs from the processor array is done by a PISO module. Fig. 4 shows interconnection of the TAGM, PISO, and dual-port memory forming an output border module, for storing the data results from the processor border. Likewise the input border module, the output border module is able to insert four data to the memory bank. Also, by replicating the output border module or relaxing one of the assumptions, it is possible to support larger or smaller processor arrays.

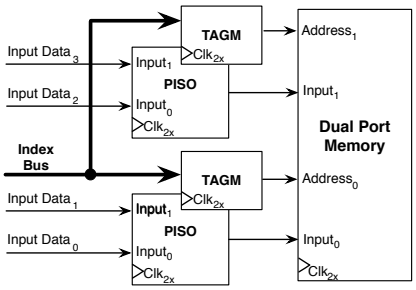

Fig. 4. Interconnection of the TAGM, dual-port memory and PISO for the output border case.

Although data produced by the processor array is recollected at the processor borders, it is not necessarily produced by the border PEs in a processor array. This situation occurs when the problem size does not fit exactly in the partitioned array. In such case the output data are produced by inner PEs and data must be sent to the processor array border, which is accomplished by placing a layer of transporting elements (TE) following the interconnection structure of the processor array. A TE has a multiplexer in charge of selecting between the data results produced by the PE or the result produced by its previous neighbor. The multiplexer output is stored in a register which delays the multiplexer output one clock cycle.

4.5 Input variable broadcast mapping

The third architectural case happens when the space-time mapping indicates that the input variable of the PRA is inserted in each PE of the processor array. After the mapping, the iteration dependent condition $\mathcal{C}^I(\vec{I})$ is mapped to time space while the indexes from vector $\vec{I}$ of the input variable are mapped to processor space. In this case the AGUs should scan one of the processor indexes by using counters. In addition, the other index is generated by replicating the AGUs at one border of the processor array. Therefore a similar approach to the input variable border case could be used for generating the memory bank addresses as well as for storing the data in dual-port memory banks. Fig. 5 shows the interconnection of the TAGM, the dual-port memory and the SIPOs interconnection in the input broadcast module. Note that there are four SIPO modules instead of two like in the case of input border case. This is because this architectural case requires a different kind of storage-interface module compared with the input border case.

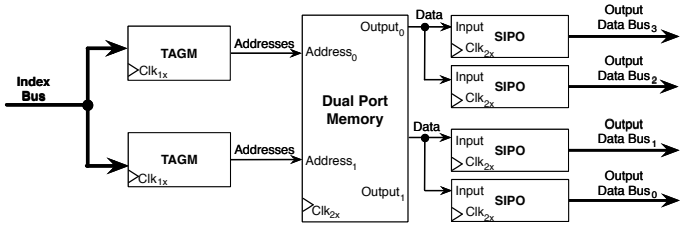

Fig. 5. Interconnection of the TAGM, dual-port memory and SIPO for the input broadcast case.

In the broadcast case, a block of data (equals to the amount of PEs), is required each time a subset of the processor space is being scanned. The data block is sent in advance to each PE where each datum of the data block is required. Such forwarding requires sending all data contained in sub-blocks simultaneously from one processor border, and sending all the sub-blocks simultaneously too. Each time that a sub-block 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 amount of registers for storing the data at each pipelining stage, and this set of registers is called broadcast data array. As consequence of the broadcast data array, at the processor array border, the SIPO requires to store SSp_x data extracted from a memory bank.

4.6 Output Variable Broadcast Mapping

The final case is when the space-time mapping indicates that the output variable of the PRA is extracted from each PE of the processor array. Like in the input variable broadcast case, after the mapping, the iteration dependent condition $\mathcal{C}^I(\vec{I})$ is mapped to time space while the indexes from vector $\vec{I}$ of the input variable are mapped to processor space. This case shares some similarities with to the input broadcast case. For instance, in both cases the

concepts of data block, pipeline stage and register array are present. However, in the output variable case since data are produced by the PEs it is required to send the data through the pipeline stages until data reach the array border. The PISO broadcast module receives a data bus of a sub-block and it selects one datum to be stored in the memory bank by using a $SSp_x - 1$ multiplexer. The multiplexer selector signal is generated by a counter which scans all data contained in the data sub-block. Finally, Fig. 6 shows the interconnection of the TAGM, the dual-port memory and a two-PISOs modules forming the output broadcast module.

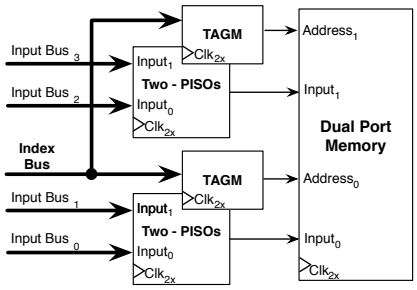

Fig. 6. Interconnection of the TAGM, dual-port memory and PISO for the output broadcast case.

5 Memory Scheme Results

5.1 Independent-Technology Results

The amount of hardware elements required by each memory architectural case in function of the processor array size (strip size), and in function of some constants c_0, c_1, n and f is shown in Table I. The SSp_x and SSp_y terms refer to any one of the two strip size parameters (SSp_0 or SSp_1), c_0 and c_1 are constant values which depends on the scheduler function. The function $g(SSp_x, SSp_y)$ is defined as shown in Eq. 3. The use of multi-port memories is represented by n while f is factor for which the memory system is faster than the processor array, respectively. Both terms are used for determining the amount of memory banks. Note that when the input border case is derived, the amount of hardware elements is linear with respect of any strip size parameters. Furthermore, if the memory assumptions of dual port memory ($n = 2$) and the twice memory operation frequency with respect of the processor array frequency ($f = 2$), then the amount of memory banks is decreased by a factor of four. Besides, if $nf > SSp_x$ then any of the two memory assumptions could be relaxed until at least $nf = SSp_x$.

$$g(SSp_x, SSp_y) = SSp_y \left(\sum_{i=1}^{SSp_x} i \right) + (SSp_x \times SSp_y) \tag{3}$$

Table I. Hardware resource utilization for each memory architectural case.

Amount of	Input Border	Output Border	Input Broadcast	Output Broadcast
Adds	$6(SSp_x)$	$6(SSp_x)$	$6(SSp_x)$	$6(SSp_x)$
Mults	$2(SSp_x + c_0)$	$2(SSp_x + c_1)$	$2(SSp_x)$	$2(SSp_x)$
Muxs	SSp_x/n	$(SSp_x \times SSp_y)\times (SSp_x/n)$	-	SSp_x
Cnts	-	-	SSp_x	$2(SSp_x)$
Regs	$2(SSp_x)$	$SSp_x \times SSp_y$	$g(SSp_x, SSp_y)$	$g(SSp_x, SSp_y)$
FFs-T	SSp_x/n	Sp_x/n	-	-
Dual-Mems	$SSp_x/(nf)$	$Sp_x/(nf)$	$Sp_x/(nf)$	$Sp_x/(nf)$

5.2 FPGA Implementation Results

This section presents the place and route (PAR) results of three memory architectural cases which are part of a complete MatMul processor array system (including control, data-path and memory). Because of the variables present in the MatMul algorithm (two input and one output variables), it is required three architectural cases, and the selection of these cases is derived by the space-time mapping. The space-time transformation was derived by using scheduler function $\vec{\lambda} = [1\ 1\ 1]$, and the projection vector $\vec{u} = [1\ 0\ 0]^t$. The three architectural cases were modeled using VHDL, placed and routed with Xilinx ISE 13.1 targeted for a Virtex-6 XC6VCX240T FPGA device. The PAR results shown in Table II correspond to the implementation of a pair of input border, input broadcast, and output border modules (six modules total) using a word size of 32-bit. Also, Table II shows the PAR results for the MatMul processor array of 8×8 PEs. Memory banks required for each architectural case and the processor array FIFOs were implemented using built-in FIFOs [2] and BRAMs. Although the results are presented for FPGA BRAMs, it is important to emphasize that off-chip DRAM could be used instead of built-in BRAMs. The processor array is able to multiply matrixes of $N \times N$ where $N < 171$ (the size of each memory bank is 512 Kbits). Note that the memory assumption of $Clk_{mem} \geq 2 \times Clk_{pa}$ is achievable since the processor array operation frequency is three times slower than the worst case for any of the memory cases. Moreover, as stated by the expressions shown in Table I, the amount of registers in the broadcast case is greater than the border modules.

In addition, Table II shows the theoretical peak bandwidth obtained by each architectural case. The input bandwidth required for the processor array is 3.72 GB/s, while the output bandwidth is 1.86 GB/s, *i.e.* the I/O bandwidth is 5.57 GB/s. Both, input and output bandwidths requirements are exceeded by the I/O bandwidth provided by the memory architectural cases due the use of several communication channels (eight channels per architectural case). The input bandwidth provided by the proposed solution is 13.24 GB/s, and the output bandwidth is 9.47 GB/s, *i.e.* the I/O bandwidth is 22.71 GB/s which is four times faster that the bandwidth required by the

processor array. This I/O bandwidth is obtained by adding each architectural case bandwidth as shown in Table II, and each one of these bandwidths is calculated by multiplying the word size times the amount of communication channels. In contrast to [11], the input bandwidth required by the 4×4 processor arrays is not satisfied by the Plesco's solution since a CPU is responsible for bringing the external data by using two communication channels. In this last case, the bandwidth required by the 4×4 array is 3.2 GB/s, while the input data transfer rate achieved is 32 MB/s due to latency in the memory controller. At the best of our knowledge, except by [11], there are not other works reporting I/O bandwidth.

Table II. PAR results for three architectural cases and a processor array implementation for MatMul.

FPGA Resources		Input	Input	Output	Processor
Name	Available	Border	Broadcast	Border	Array
Slices Registers	301,440	591	2,078	205	14,610
Slices LUTs	150,720	459	781	522	12,194
RAMB36E1	416	29	29	29	-
FIFO36E1	416	-	-	-	16
DSP48E1	768	16	16	16	192
Max. Frequency (MHz)		227.17	186.43	296.03	57.98
Peak Bandwidth (GBytes/s)		7.27	5.97	9.47	5.57

6 Conclusions

An external memory interfacing system for inserting/extracting data to/from the processor array derived from a PRA has been presented. The memory system is composed by four architectural memory cases based on the use of a GALS approach, and the use of dual-port memory banks. By relaxing the GALS assumption and/or replacing the dual-port memory by single ones, it is possible to fit the memory system to smaller or slower processor arrays. The place and route results show that the assumption made of the external memory working in a faster clock than the processor array is achievable, since the processor array clock frequency is three times slower than the worst case of any architectural memory cases. Besides, the bandwidth achieved by the complete system fulfills the bandwidth required by the processor array. Other algorithms such as one-dimensional convolutions, system equations solvers, and matrix decomposition (like Cholesky, LU or QR) modeled as PRA are suitable of being implemented as processor arrays using the proposed memory scheme.

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. Guzmán for his contribution to this research.