Hardware reduction using a 6-connectivity interconnection network over a 4-connectivity VLSI asynchronous array processor

Valentin Gies
ENSTA
32, Bd Victor
75015 Paris, FRANCE
Email: contact@vgies.com

Thierry M. Bernard
ENSTA
32, Bd Victor
75015 Paris, FRANCE

Alain Mérigot
Institut d'électronique fondamentale
Université Paris Sud
Orsay, FRANCE

***Abstract*— Low level local image processing is efficiently performed by array processors operating in SIMD mode. Performing mid-level regional image processing leads to use local combinatorial operators combined with an asynchronous programmable interconnection network. However, this approach has an important hardware cost because asynchronism implies the use of combinatorial operators with many inputs. This cost should be reduced for a dense VLSI implementation. To do so, we propose to increase the connectivity level of the interconnection network as a mean to use only 2-input asynchronous combinatorial operators. Results are presented on the example of the regional sum mid-level primitive in vision chips. An extension of the methodology to higher connectivity levels is then proposed.**

I. Introduction

With the evolution of multimedia and embedded technologies such as personal digital assistants (PDAs), mobile phone or embedded systems for robotic applications, image processing and analysis have to be performed on fast and low power vision chips. In this paper we focus on VLSI array processors for image processing. One of the most attractive features of such chips is that, compared with standard vision systems, power consumption is reduced by 2 to 3 orders of magnitude.

Using an array processor in SIMD mode is very efficient for regular low-level local computations [1].However, to carry on higher levels of image processing, array processors have to deal with mid-level irregular arbitrary sized and shaped image data, such as regions. To do so, fast communications and computations over data dependant regions are necessary. In contrast with neighbor-to-neighbor communications used in SIMD array processors, regional operators require communication between sparse and distant pixels. Programmable neighbor-to-neighbor connections [2] allow to implement data-dependant communication networks within the SIMD framework, but with poor synchronous performances : communication speed is limited to "one pixel farther per clock cycle" [2]. This leads to use asynchronous instead of synchronous communication. Thus array processor have to feature programmable connections and asynchronous communications and computations to efficiently support mid-level vision. However, while low-level synchronous SIMD operators can be made simple yet versatile, mid-level asynchronous operators have a much higher hardware cost. The reason is that they must include combinatorial logic to simultaneously handle several data in each elementary processor of the processor array, in particular 5 data in 4 connectivity computations.

In the present paper, the regional sum is considered as the exemplary regional asynchronous operator. Section II recalls existing asynchronous implementations of the regional sum operator, based on a dedicated adder in each elementary processor. In section III, we propose to use an inter-connection network in 6-connectivity for lowering the hardware cost of asynchronous operators. The trick is that we manage to content ourselves with operators having a minimum number of inputs, actually 2 or 3 inputs. Algorithm for installing the interconnection network is presented, a hardware cost comparison is proposed. In section IV, an extension to the "regional sum" over a 6-connectivity array processor connected with a 8-connectivity interconnection network is also proposed.

II. An asynchronous regional operator : The regional sum

In this section, asynchronous regional sum is considered as an example of regional mid-level image processing primitive. Computing a regional sum implies to collect data from all the pixels of the region. Collecting and adding these data in one chosen place implies moving data on long distances. In order to cope with this problem, data must be added locally and forwarded. To do so, a possibility is to connect pixels into a direct acyclic graph structure [3] also called a spanning tree, with a combinatorial adder in each pixel. Each operator add the binary values provided by the neighboring preceding pixels in the graph, and a local value. This solution has been proposed and implemented by in [4] using dynamically reconfigurable 1-D chains of pixels set by external programming. Limitations of such an implementation (in a 1-D network, it is impossible to cover an arbitrary shaped and sized region) have been coped with in a 2-D implementation, using a dynamically self-configurable 2-D tree interconnection network over a region

[5] [6]. Unfortunately, this latter implementation have a very important hardware cost

To highlight the reasons of this important cost, and to reduce it, we analyze the regional sum algorithm. First, the sum algorithm over a tree is described, then the spanning tree installation is explained, and finally the hardware cost corresponding to each function is calculated.

A. Sum algorithm

An asynchronous multi-input local adder is present in each pixel. In each processor, adder inputs are connected to a local binary data bit and to the preceding local adders less significant output bit (usually called the sum bit). The less significant output bit of the sum computed by the local adder can be interpreted as the parity of the number of 1's inputs of the adder in that pixel. By associativity of the parity operator, the result in the root of the spanning tree structure is the parity of the number of 1's among all local binary data bits of the tree. This value is also the least significant bit of the sum of local binary data of the whole region.

At the beginning of the regional sum algorithm, the least significant bit of the operand in each pixel is placed in the local binary data bit. The first step is to run the global combinatorial sum computation and to get the least significant bit of the sum in the adder in the root, this bit is the less significant bit of the sum. The second step is to synchronously memorize in each pixel the carries of each local sum. Then, the process is iterated to produce each bit of the sum. Result is located on the tree root.

Such an algorithm computes the regional sum in N asynchronous operations where N is the number of bits required to represent the sum. These combinatorial operations are executed in a synchronous sequence. To compute the sum in 4-connectivity, a 4-input adder in each pixel is needed (3 neighbors and 1 local data bit).

B. Asynchronous spanning tree installation

As explained above, a spanning tree is necessary for the regional sum computation. It cannot be settled efficiently in a synchronous way, because the number of steps of the algorithm would grow linearly with the geodesic diameter, and it would take a long time. So, this task has to be performed in an asynchronous way.

The asynchronous algorithm used is the following one. At initialization, all connections between pixels of a same region are established. All pixels are inactive and a root is chosen, deterministically or at random. Then the root is activated, and communicate its state to the neighbor pixels. Each activated pixel keeps in memory the connection threw which it was activated, and forward its active state to its neighbors. This process propagates asynchronously throughout the region until all pixels are active. The oriented spanning tree is obtained looking at the unique connections used for the activation of each pixel.

As explained before, a spanning tree is a direct acyclic graph, that means each pixel must have only one antecedent. During the algorithm, a pixel may have to choose between 2 or more antecedents if they want to activate the considered pixel at the same time. For this reason, in 4-connectivity a 4-input arbiter is needed in each pixel.

C. Tree bit-serial adder asynchronous hardware cost

The different asynchronous components needed to perform regional sum computation and spanning tree installation have been defined before. Asynchronous part of each elementary processor is composed of a 4-input arbiter (32 transistors), a 4-input adder(44 transistors), and 6 programmable connections (necessary for choosing 3 out of 4 inputs). The hardware cost of the asynchronous part is finally 82 transistors. Such an important cost is worth being reduced for a VLSI implementation.

III. Network topology for regional sum computation

Asynchronous dedicated operators used for sum computation are expensive mostly because they have 4 inputs. To cut this transistor expense, a lighter structure is desirable. When looking at an example of computing network, we notice that most of the cells exploits their 4-input convergent operators as simple 1-input operators only. This is a waste of resources. Is there any way to better distribute the network, that allows the use of k-input convergent operators with k smaller than 4 ?

First, let's recall that $k = 1$ corresponds to 1-input operators and is therefore insufficient. What about using a 2-input asynchronous operator? Let's call it a *2-input convergent operator*. Let's consider a computation network over a region with m pixels. Connecting these m pixels together in a tree structure requires exactly $m - 1$ operators, convergent or not. How to settle them among m pixels ? Only one 2-input convergent operator in each pixel could be enough. There are 2 main issues for implementing such a structure. The first one is the network topology needed to implement it, the second one is how to install the spanning tree.

A. Network topology

We recall that a 4-connectivity network combined with the use of 2-input convergent operators is insufficient to set-up a network over an arbitrary shaped region. Let's consider a cross-shaped region of 5 pixels, such a region is an example of this impossibility (Fig. 2). A solution is to increase the connectivity level used. An 8-connectivity interconnection network could be an obvious solution to the problem, allowing vertical, horizontal and both diagonal connections. However, the hardware cost would be rather expensive. A better solution is to use 6-connectivity and 2-input convergent operators. First, let's show that this solution fits our needs. For this, let's consider the pixel matrix as a hexagonal mesh (Fig. 1). Thanks

Fig. 1. Transformation from square to hexagonal mesh

to hexagonal mesh properties, an arbitrary pixel configuration can be connected with only 2-inputs operator. For example, installation of a spanning tree using only 2-input operators over a 5 pixels cross-shaped region, one of the configurations impossible to map in 4-connectivity, is proposed (Fig. 2). 6-connectivity is the lowest connectivity level allowing the

Fig. 2. Spanning tree over a cross-shaped region

connection of an arbitrary shaped region into a tree structure with only binary operators, and it leads to the lowest possible hardware cost using asynchronous computation operators.

B. Asynchronous spanning tree installation

Using 4-inputs operators, installation of a spanning tree is a rather simple task. Starting from a fully connected network, a signal propagates from the tree root threw the network until all pixels have been reached (cf. II-B). Using 2-inputs operators, this task is much more difficult because at the initialization of the algorithm, each pixel can be connected to 2 pixels only, and not to all its neighbors. Fortunately, connecting all the neighbors is something simplifying the construction of the spanning tree but not really necessary. What is only needed is to be sure that propagation starting from one pixel will reach all the other pixels of the region. That means every pixel of the region has to be connected to all other pixels. Such a region is called a strongly connected component (SCC). The issue is how to build a SCC in 6-connectivity using only 2-inputs operators.

1) Algorithm principles: A way to solve this problem is to connect all the boundary pixels of the region into a clockwise oriented chain and then to connect all the pixels not connected yet and the boundary rings together. Fig.3 shows the original region on the top, and its corresponding hexagonal representation after the initialization of the connections on the bottom. As explained before, pixel inputs are connected to a maximum of 2 other pixels.

According to this connection method, boundary pixels are

Region being connected

Corresponding hexagonal region with connections initialized

Fig. 3. Example of spanning tree initialization on a 6-connectivity network using only 2-inputs operators

connected in a SCC (a ring is a simple SCC), and other pixels are added to the SCC thanks to bi-directional connections, this ensuring them to be part of the SCC.

The final step of the proposed method is to extract a spanning tree from the SCC by propagating a token from the root as explained before in section II-B. We will not come back on this last point here.

2) Algorithm for connecting a SCC: The algorithm used is very simple, and can performed in a very cheap and fast synchronous way. Initialization of the connections can be done by only considering 6 local pixel configurations as described in Fig. 4. The pixel to connect is double-circled. Black pixels are pixels belonging to the region while white ones are pixels outside the region.

An important fact is the non-isotropy of the local transformation. This is a consequence of mapping a 4-connectivity square mesh with an hexagonal network. The diagonal connection, not present in 4-connectivity, is here used for establishing all the non-boundary connections.

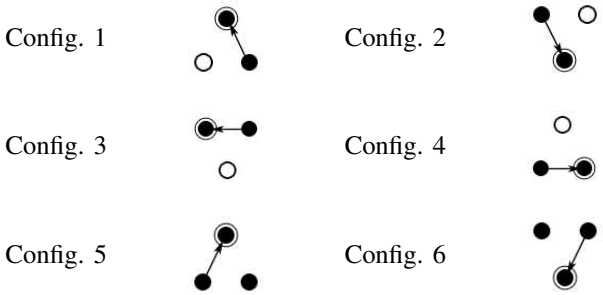


Fig. 4. Local configuration for SCC initialization

3) Validity of the algorithm: Having presented the principles and operation of the algorithm, let's demonstrate its validity. By construction, all pixels are connected in a same SCC. The only point to verify, is to be sure that with this method, pixel inputs can be connected to no more than 2 neighbor pixels, this allowing using 2-inputs operators.

As shown in Fig.4, each configuration sets-up one input connection. We have to verify that if a pixel neighborhood corresponds to 2 configurations, all the other configurations are false. For this let's consider the mutual exclusions of the configurations. Configuration 1 excludes configuration 3 and 5. Configuration 3 excludes 1 and 5, and configuration 5 excludes 1 and 3. Finally only one odd-numbered configuration can be true at one time. It is the same for even configuration. Only one even configuration can be true for a given neighborhood. There are no exclusions between odd and even configurations. Finally, a pixel neighborhood can only match one even and one odd configuration. That means a maximum of 2 configurations can be valid at one time, and maximum 2 input connections will be set-up in the pixel.

4) Performance of the algorithm: The proposed algorithm for initializing the SCC can be performed very efficiently in a synchronous non-iterative way. This allows using this algorithm on a massively parallel synchronous machine having only limited resources for synchronous computation.

C. Hardware reduction

Using a 6-connectivity interconnection network over a 4-connectivity array processor allows to reduce the hardware cost dedicated to asynchronous regional sum to one 2-input arbiter (8 transistors) and one 3-inputs adder (20 transistors) in each pixel. 3 inputs are necessary for the adder : one for the local bit, and two for the neighbor connections. However, the necessary number of programmable connections increases. 2 out of 6 neighbors have to be connected at one time. Consequently, the minimal number of programmable connections necessary is 5 connections for each input. Finally, the number of transistors needed is $8+20+10 = 38$ transistors. 44 transistors are saved in each pixel by using a 6-connectivity topology. This leads to a reduction of 54% of the hardware expense. However, the algorithmic capabilities are remaining the same.

IV. Network topology for regional sum computation over a 6-connectivity region

Increasing the connectivity level of the interconnection network for reducing hardware cost of regional computation using only 2-input convergent operators can be extended to 6-connectivity array processors. A 8-connectivity interconnection network is used. The interconnection network will be represented in a squared mesh because as shown in Fig. 5, in a squared mesh, a 8-connectivity neighborhood is more regular (central symmetry) than in a hexagonal mesh (only 2 axial symmetries).

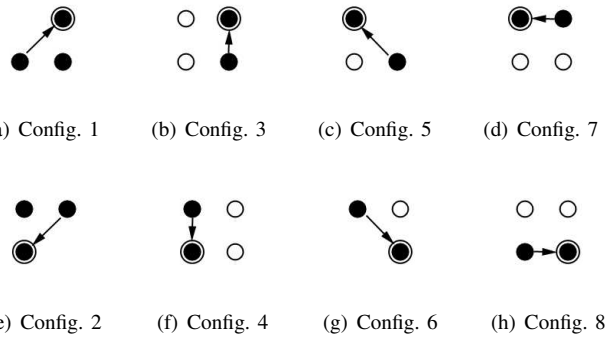
The principles of the algorithm remain the same as in 4-

Fig. 5. Transformation from squared to hexagonal 8-connectivity mesh

connectivity. Boundary pixels are connected into a clockwise oriented chain and non-boundary pixels are connected to the SCC formed by the boundary ring using a synchronous algorithm (cf. Fig. 6).There are 8 configurations to consider for setting the interconnection network. All theses configurations are used for connecting boundary pixels. Configurations 1 and 2 are also used for connecting non-boundary pixels to the boundary ring. With the proposed algorithm, an 6-connectivity network can be connected using 2-input convergent operators.

A. Hardware reduction

Using an 8-connectivity interconnection network over a 6-connectivity array processor allows to reduce the hardware cost dedicated to asynchronous regional sum to one 2-input arbiter (8 transistors) and one 3-inputs adder (20 transistors) in each pixel. 3 inputs are necessary for the adder : one for the local bit, and two for the neighbor connections. Considering interconnections, 2 out of 8 neighbors have to be connected at one time. Consequently, the minimal number of programmable connections necessary is 7 connections for each input. Finally,



(a) Config. 1 (b) Config. 3 (c) Config. 5 (d) Config. 7

(e) Config. 2 (f) Config. 4 (g) Config. 6 (h) Config. 8

Fig. 6. Local configurations for SCC initialization in 6-connectivity using a 8-connectivity interconnection network

the number of transistors needed is $8+20+14 = 42$ transistors. Using 7-input asynchronous adders and 6-input arbiters leads to an hardware cost of 156 transistors (72 for the adder and 84 for the arbiter). Finally, 114 are saved in each pixel by using a 8-connectivity interconnection network. This leads to a drastic reduction of 73% of the asynchronous hardware expense, without lowering algorithmic capabilities.

V. Conclusion

We recalled implementations for regional sum computation through regions of an the image. After evaluating the hardware cost of dedicated operators needed for computing the sum asynchronously, we proposed using a 6-connectivity interconnection network over a 4-connectivity array processor or a 8-connectivity interconnection network over a 6-connectivity array processor for reducing the hardware cost of asynchronism using only 2-input operators. This leads to a important reduction of more than half the original transistor cost in 4-connectivity. This reduction becomes drastic in a 6-connectivity array processor. Such a reduction offers an opportunity for achieving a dense large scale implementation of this circuit. We expect to realize this implementation very soon.

References

[1] F. Paillet, D. Mercier, and T. Bernard, "Second generation programmable artificial retina," in *IEEE ASIC/SOC Conf.*, Sept. 1999, pp. 304–309.

[2] H. Li and Q. Stout, *Reconfigurable Massively Parallel Computers*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[3] B. Ducourthial and A. Merigot, "Graph embedding in the associative mesh model, Tech. Rep. TR-96-02, 1996. [Online]. Available: citeseer.ist.psu.edu/ducourthial96graph.html

[4] T. Komuro, S. Kagami, and M. Ishikawa, "A dynamically reconfigurable simd processor for a vision chip," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 1, pp. 265–268, 2004.

[5] A. Merigot, "Associative nets: A graph-based parallel computing net," *IEEE Transactions on Computers*, vol. 46, no. 5, pp. 558–571, 1997.

[6] D. Dulac, S. Mohammadi, and A. Merigot, "Implementation and evaluation of a parallel architecture using asynchronous communications," in *CAMP*, 1995, pp. 106–111.