Performance Optimization on Low-Cost Cellular Array Processors

Alberto Broggi*
Dipartimento di Ingegneria dell'Informazione
Università di Parma
Parma, ITALY, I-43100

Abstract

A massively parallel architecture is composed of a high number of processing elements (PE), but seldom a 1:1 mapping between the PEs and the data set can be achieved. To overcome this problem, a processor virtualization mechanism is needed.

This work provides a theoretical study on performance optimization in the specific virtualization process used when a small amount of memory is associated to each PE. Moreover, this paper presents an algorithm for performance optimization and for the determination of the maximum processing speed on low-cost cellular array processors implementing the above mentioned virtualization mechanism. These considerations are then used to improve the execution of morphological image processing tasks on the first hardware prototype of the special-purpose cellular array processor PAPRICA.

1 Introduction

Two virtualization mechanisms are generally implemented on massively parallel systems: the choice depends mainly on the amount of memory associated to each processing element (PE). In the first one [12, 7, 13, 8, 9] the computation is serialized within each processor, which contains several data; conversely, in the second case [16, 14, 15, 6], namely when each PE can only handle a small amount of memory, the computation is serialized in windows: the Processing Array (PA) is loaded with a sub-window of the data set, then the computation is performed until a special instruction (hereinafter referred to as **UPDATE**) is reached, and finally the result is stored back into the memory. These steps are iterated until all the sub-windows have been processed; then the first sub-window is reloaded and the computation is resumed until the next **UPDATE** instruction is found. This second solution, generally implemented on low-cost systems and analyzed in detail in [5, 11], implies the use of an external memory for data storage.

The position and the frequency of **UPDATE** instructions affect directly the processing speed of the system. Their optimum positioning within the sequence of the program instructions is a task that requires an attention that is comparable to the attention needed to develop the algorithm itself.

The main problem with this implementation of virtual processors is mainly due to the limited dimensions of the PA, where the border processors can't access their complete neighborhood. In fact, after the execution of an instruction which implements a 3×3 morphological function [17], the values stored in the border processors of the PA are meaningless. Thus only a portion (hereinafter referred to as the *Validity Area*) of the data processed by the PA will hold significative data. Thus the next windows that will be transferred into the PA will be partially overlapped with the previous ones, in order to correctly evaluate the previously invalidated results.

In the next section a theoretical analysis of the performances is derived for a generic cellular architecture implementing the above mentioned virtualization mechanism. Since the main target of this study is the speed-up of applications [1, 2] written for PAPRICA special-purpose array processor [3, 4, 10], section 3 provides an optimization method for the automatic positioning of **UPDATE** instructions. Section 4 ends the paper with some PAPRICA performance figures together with some concluding remarks.

2 Theoretical performance evaluation

When dealing with image processing tasks, the processing speed S_{pr} is defined as the number of pixels

*This work was partially supported by CNR Progetto Finalizzato Trasporti under contracts 93.01813.PF74 and 93.04759.ST74. The author can be reached at the following e-mail address: broggi@CE.UniPR.IT

0-8186-6322-7/94 $04.00

processed over the total processing time for a given sequence of instructions:

$$S_{pr} \triangleq \frac{\text{Number\_of\_pixels\_processed}}{\text{Processing\_time}} . \quad (1)$$

The ideal case:

The maximum processing speed can be obviously reached when the PA can contain the complete image or when the sequence of instructions does not reduce the validity area. In this ideal case, the number of pixel processed is the number of pixel in the full image ($F \times F$), while the *processing time* t is the sum of the time needed

(a) to load the image into the PA,

(b) to run the application, and

(c) to save back the data into the image memory.

The *loading time* and the *saving time* can be assumed equal and given by the number of elementary transfer operations (Q^2, where Q is the PA linear dimension) time the basic transfer operation duration (T_M). The *execution time* is given by the number of instructions (L) time the instruction duration (a single clock cycle, T_C) and therefore:

$$t = 2\,Q^2\,T_M + L\,T_C . \quad (2)$$

The processing speed is then given by:

$$S_{pr} = \frac{Q^2}{2\,Q^2\,T_M + L\,T_C} . \quad (3)$$

Assuming that T_C and T_M are of the same order of magnitude, if L is negligible with respect to $2\,Q^2$, the previous expression reduces to:

$$S_{pr} \simeq \frac{Q^2}{2\,Q^2\,T_M} = \frac{1}{2\,T_M} . \quad (4)$$

Equation (4) shows that when the *execution time* is negligible (in the ideal case) the total *processing time* is approximately given by the I/O time: a problem, whose scalar version was CPU-bounded, is now reduced, through the use of a massively parallel system, to an I/O-bounded problem.

The real case:

Let's consider now the case in which the number of PEs is less than the number of image pixels and assume that the program contains some instructions responsible of the reduction of the validity area. The parameter G, associated to each instruction, measures the reduction of the validity area: figure 1 shows a few typical instructions and their associated G factor.

The aim of the **UPDATE** instructions is to avoid that the validity area is reduced to zero. It is obvious that if the frequency of the **UPDATE** operations increases, the *processing time* grows up because of the time needed to transfer the same windows from the memory to the PA; similarly, the *processing time* grows up also when the number of **UPDATE** operations decreases too much because most of the processing is meaningless. It follows that there is an optimum value for the **UPDATE** frequency that maximize the processing speed.

Assuming the dimensions of the image ($F \times F$) greater than the dimensions of the PA ($Q \times Q$), the processing time can be computed as the sum of the processing time of each single window (t_{1w}) time the number of windows to be processed (n_w): $t = n_w\,t_{1w}$.

The number of windows processed is not simply $\frac{F^2}{Q^2}$ because, due to the reduction of the validity area, the windows must be partially overlapped. n_w is in fact expressed as:

$$n_w = \frac{F^2}{Q'^2} , \quad (5)$$

where the *effective linear window dimension* Q' is given by:

$$Q' = Q - \frac{2\,G}{n_{upd}} . \quad (6)$$

The ratio $\frac{2\,G}{n_{upd}}$ denotes the average linear validity area reduction due to a program formed by a set of instructions with a reduction factor G, interleaved with n_{upd} **UPDATE** operations.

Some special cases can show the plausibility of the previous expressions:

- If $G = 0$, Q' reduces to Q (figure 2.a);
- If $G > 0$ and $n_{upd} = 1$ (one **UPDATE** at the end of the program), $Q' = Q - 2\,G$ (figure 2.b);
- If $G > 0$ and $n_{upd} = 2$, (two **UPDATE** operations), $Q' = Q - 2\,\frac{G}{2}$ (figure 2.c).

The total processing time of each single window (t_{1w}) is then given by the number of **UPDATE** blocks time the duration of the elaboration between two **UPDATE**s. The *elaboration time* is thus given by the sum of the read and write times, plus the execution time of the $\frac{L}{n_{upd}}$ instructions in the average **UPDATE** block. Thus the *processing time* is given by:

$$t = n_w\,t_{1w} , \quad (7)$$

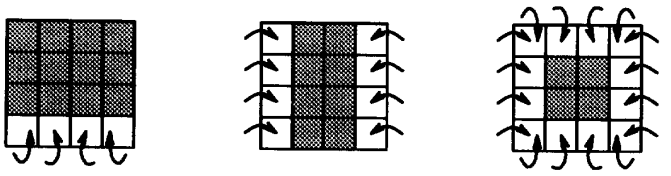

Figure 1: The G factor and, in gray, the Validity Area after the execution of the specific instruction.

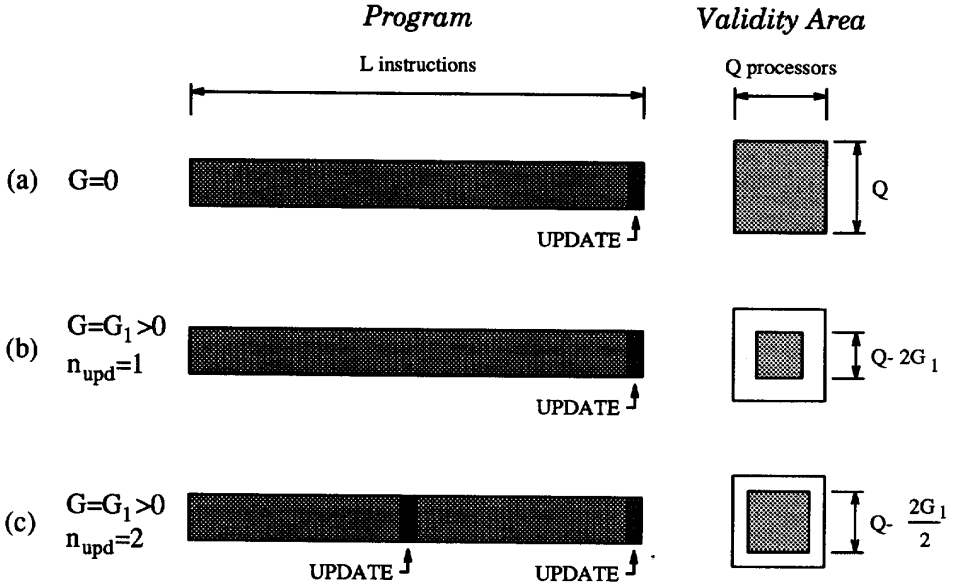

Figure 2: Validity area reduction at the end of each **UPDATE** block.

which becomes:

$$t = \left[\frac{F^2}{\left(Q - \frac{2\,G}{n_{upd}}\right)^2}\right] \times \left[n_{upd}\left(2\,Q^2\,T_M + \frac{L}{n_{upd}}\,T_C\right)\right]. \tag{8}$$

After few algebraic manipulations equation (1) becomes

$$S_{pr} = \frac{\left(Q - \frac{2\,G}{n_{upd}}\right)^2}{2\,Q^2\,T_M\,n_{upd} + L\,T_C}. \tag{9}$$

Note that equation (3) can be obtained from the previous one when $n_{upd} = 1$ (a single **UPDATE** at the end of the program) and $G = 0$ (as in the ideal case).

The processing speed is then a function of parameters that depend:

1. on the system technology and architecture (T_M, T_C, Q),
2. on the specific computational task (L, G), and
3. on the number of **UPDATE** instructions (n_{upd}).

As far as $L\ T_C$ is negligible with respect to $2\ Q^2\ T_M\ n_{upd}$ - for example when Q is sufficiently large - the processing speed does not depend on the length of the program. Moreover, since the system parameters (Q, T_M, T_C) are fixed by the physical architecture, and the G factor is determined by the specific application, the processing speed can be tuned by changing the number and position of **UPDATE** instructions.

3 Performance optimization

In order to get the optimum value for n_{upd}, let's consider, for sake of simplicity, the case in which $2\,Q^2\,T_M\,n_{upd} \gg L\,T_C$. In this case equation (9) can be approximated by removing the term $L\,T_C$. Thus, the maxima of S_{pr} can be found in the solutions of the following derivative:

$$\frac{\partial}{\partial\, n_{upd}}\, S_{pr} \simeq \frac{\partial}{\partial\, n_{upd}}\, \frac{\left(Q - \frac{2\,G}{n_{upd}}\right)^2}{2\,Q^2\,T_M\,n_{upd}} = 0\,, \qquad (10)$$

which are:

$$\begin{cases} x_1 = 2 \implies \left(n_{upd}\right)_1 = 2\,\frac{G}{Q} \\ x_2 = 6 \implies \left(n_{upd}\right)_2 = 6\,\frac{G}{Q} \end{cases} \qquad (11)$$

The diagram presented in figure 3 (S_{pr} vs x, where x is defined as $x \triangleq \frac{Q\,n_{upd}}{G}$) shows that the first solution corresponds to a minimum while the second one identifies a maximum.

The meaningful range for x is given by: $2 < x \leq \frac{Q\,L}{G}$, where the limit values refer to:

- $x = 2 \implies Q\,n_{upd} = 2\,G$:
  in this case the validity area is completely eroded and S_{pr} reduces to 0.
- $x = \frac{Q\,L}{G} \implies n_{upd} = L$:
  in this case each instruction is followed by an **UPDATE** operation. $n_{upd} > L$ would lead to a program with two or more adjacent **UPDATE** instructions which has no meaning.

The optimum value of n_{upd} is then given by:

$$\left[n_{upd}\right]_{opt} = 6\,\frac{G}{Q}\,, \qquad (12)$$

and it allows to estimate an upper bound for the processing speed.

To simplify the programming activity an algorithm has been developed to introduce properly the **UPDATE** instructions in an assembly program with the aim of maximizing its execution speed.

3.1 Analysis of the algorithm

Considering the real case ($G > 0$), if the program does not contain any implicit **UPDATE**[1], the algorithm is simply based on the scanning of the assembly program, in order to compute the cumulative value of the validity area reduction $g(x)$ as a function of the specific sequence of instructions $\ell(x)$, $x = 0, ..., L$. Whenever the value of $g(x)$ reaches the value $\frac{Q}{6}$ (obtained from equation (12) when $n_{upd} = 1$), an **UPDATE** instruction is inserted and the value of $g(x)$ is reinitialized to 0. From easy considerations it follows that an **UPDATE** instruction is positioned when approximately $\frac{5}{9}$ of the processors hold meaningless data.

If the program contains implicit **UPDATE** instructions, it is possible to consider the sections between two **UPDATE**s as independent programs. The problem is thus reduced to the case previously discussed.

[1] It has to be noted, that an implicit **UPDATE** instruction is generally included in some control-flow constructs (for example in the PAPRICA system the REPEAT UNTIL, FOR ENDFOR and IF ELSE ENDIF constructs).

4 Conclusions

Following the expressions derived in the previous sections, the processing speed becomes:

$$S_{pr} = \begin{cases} \frac{Q^2}{2\,Q^2\,T_M + L\,T_C} \simeq \frac{1}{2\,T_M}\,, & \text{if } G = 0; \\ \frac{\left(\frac{2}{3}Q\right)^2}{12QT_MG + LT_C} \simeq \frac{Q^2}{25QT_MG + 2LT_C}\,, & \text{if } G > 0. \end{cases} \qquad (13)$$

Using the set of system parameters reflecting the current PAPRICA hardware prototype (Q^2=256, T_C=500 ns, and T_M=250 ns), together with a typical set of values for the application-dependent parameters ($L \simeq 200 \div 300$ and $G \simeq 10 \div 20$ [2]), in the real case the processing speed becomes $S_{pr} \simeq 100 \div 200$ kPixel/s, while in the ideal case it reaches its maximum: $S_{pr} =$ 2 MegaPixel/s.

The main result of this work is the determination of an automatic partitioning of a single application, which requires a lot of accesses to the neighboring processors, into different segments of code with the aim of maximizing the processing speed. Following the results obtained using the proposed optimization, the processing speed becomes a function of only the system parameters and the specific computational task.

References

[1] G. Adorni, A. Broggi, G. Conte, and V. D'Andrea. A self-tuning system for real-time Optical Flow detection. In *Proceedings IEEE System, Man, and Cybernetics Conference*, volume 3, pages 7–12, Le Toquet, France, October 17-20 1993. IEEE System, Man and Cybernetics.

[2] A. Broggi. Parallel and Local Feature Extraction: a Real-Time Approach to Road Boundary Detection. *IEEE Transactions on Image Processing*, February 1995. In press.

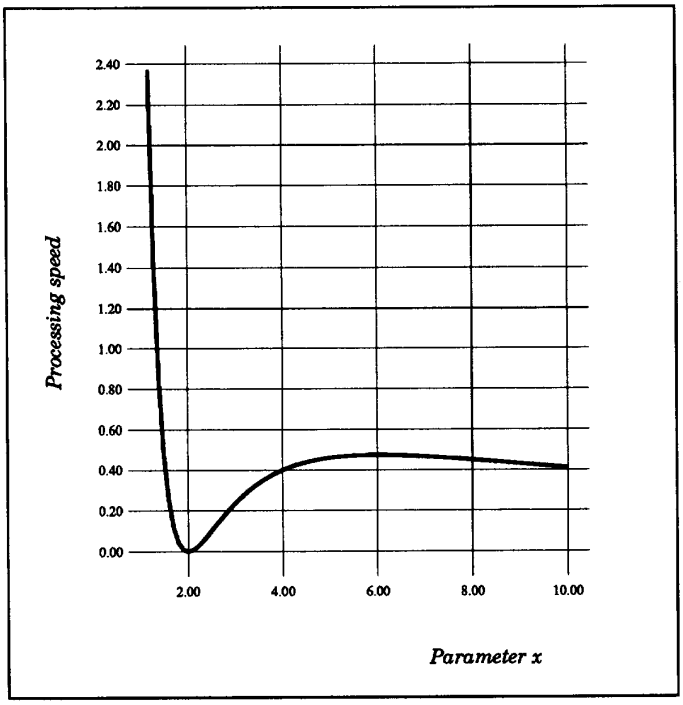

Figure 3: Processing speed S_{pr} versus $x = \frac{Q\ n_{upd}}{G}$

[3] A. Broggi, G. Conte, F. Gregoretti, L. Reyneri, L. Rigazio, C. Sansoè, and C. Zamiri. PAPRICA. In *CAD and Architectures: Reports on Architectures and Algorithms for VLSI Design*, pages 21–141. CNR - Progetto Finalizzato MADESS, Roma, 1990.

[4] A. Broggi, V. D'Andrea, and F. Gregoretti. A low-cost parallel VLSI architecture for low-level vision. In M. Takagi, editor, *MVA'92 - IAPR Workshop on Machine Vision and Applications*, pages 11–15, Tokyo, Japan, 1992. International Association for Pattern Recognition, IAPR.

[5] A. Broggi, S. Mora, and C. Sansoè. Enhancement of a 2D Array Processor for an Efficient Implementation of Visual Perception Tasks. In M. A. Bayoumi, L. S. Davis, and K. P. Valavanis, editors, *Proceedings CAMP'93 - Computer Architectures for Machine Perception*, pages 172–178, New Orleans, December 15-17 1993. IEEE Computer Society.

[6] G. Conte, F. Gregoretti, L. Reyneri, and C. Sansoé. PAPRICA: a Parallel Architecture for VLSI CAD. In A.P.Ambler, P.Agrawal, and W.R.Moore, editors, *CAD Accelerators*, pages 177–189. North Holland, Amsterdam, 1991.

[7] M. Duff. Parallel Processors for Digital Image Processing. In P. Stucki, editor, *Advances in Digital Image Processing*, pages 265–279. Plenum Press, London, 1979.

[8] T. Fountain and V. Goetcherian. CLIP4 Parallel Processing System. *IEE Proceedings*, 127E:219–224, 1980.

[9] T. Fountain and K. Matthews. The CLIP 7A Image Processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):310–319, May 1988.

[10] F. Gregoretti, L. M. Reyneri, C. Sansoè, A. Broggi, and G. Conte. PAPRICA - A Dedicated Processor for Morphological Image Analysis. In *Proceedings 7th IAPR International Conference on Image Analysis and Processing*, Bari, Italy, September 20-22 1993.

[11] F. Gregoretti, L. M. Reyneri, C. Sansoè, A. Broggi, and G. Conte. The PAPRICA SIMD array: critical reviews and perspectives. In L. Dadda and B. Wah, editors, *Proceedings ASAP'93 - IEEE Computer Society International Conference on Application Specific Array Processors*, pages 309–320, Venezia, Italy, October 25-27 1993. IEEE Computer Society - EuroMicro.

[12] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, Ma., 1985.

[13] M.J.Duff, D.M.Watson, T. Fountain, and G. Shaw. A cellular logic array for image processing. *Pattern Recognition*, 15:229–247, 1973.

[14] NCR Corporation, Dayton, Ohio. *Geometric Arithmetic Parallel Processor*, 1984.

[15] S. Reddaway. DAP - A distributed array processor. In *1st Annual Symposium on Computer Architectures*, pages 61–65, Florida, 1973.

[16] L. A. Schmitt and S. S. Wilson. The AIS-5000 Parallel Processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):320–330, May 1988.

[17] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, London, 1982.