

Word Parallelism vs Spatial Parallelism: a performance optimization technique on the system

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

Abstract

Starting from the analysis of the hardware efficiency of SIMD systems which use an external memory for data storage, this paper discusses a critical point in hardware design. In particular it presents a technique aimed to the maximization of the data bus efficiency. This technique is based on the transformation of the initial data set into a packed one, and can be successfully implemented on systems which allow a dynamic mapping between the Processing Array and the External Memory. As an example, the PAPRICA system is considered.

An optimizing Assembly-to-Assembly translator has been developed for the automatic conversion of a generic PAPRICA Assembly program working on binary data sets into its equivalent version working on packed data sets. An example of a thinning filter shows how this technique can improve the performances.

1 Introduction

Many specific tasks of Computer Vision systems are very well suited to massively parallel architectures due to their fine grain parallelism and to the peak processing power requested by real-time constraints. However sometimes these processing structures are to be embedded into larger systems whose requirements of cost, power and size do not match with those of commercially available machines.

On the other hand each task does not necessarily require the whole set of processing and communication facilities of a general purpose massively parallel machine and therefore application specific solutions with limited complexity must be designed and optimized.

*This work was partially supported by CNR Progetto Finalizzato Trasporti under contract number 91.01031.PF93. The author can be reached at the following e-mail address: broggi@CE.UniPR.IT

This paper analyzes a key point in the implementation of SIMD massively parallel low-cost systems characterized by a low amount of memory owned by each Processing Element (PE): these systems use an external memory for data storage. This work focuses on the tuning of the parallelism of the data-bus which links the Processor Array (PA) with the external memory (Image Memory, in Image Processing applications). As an example, the PAPRICA system [4] is considered.

The logical organization of the external memory is a critical design issue, and two possible solutions can be devised, as shown in figure 1. Assuming that the

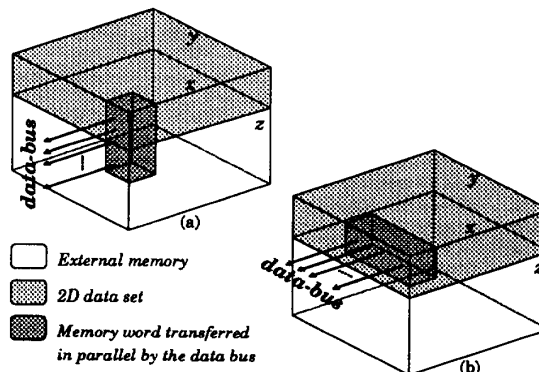


Figure 1: Two different solutions for the mapping of a memory word within the image memory

2D data set lays in the (x, y) plane, in the former (figure 1.a) the memory words lay in the z direction, orthogonal to the data set plane. In the latter the memory words lay in one of the x or y directions. The former has the disadvantage that for each data transfer a fixed number of bits (equal to the bus parallelism p_{BUS}) are moved from the image memory to the PA, regardless to the application requirements. As a con-

sequence, the data bus efficiency seldom reaches high values, because the parallelism of the data set seldom matches the bus parallelism. Conversely, the latter (figure 1.b) has the advantage of moving through the data bus only the required amount of information, but it requires some additional hardware extensions. In fact, in the case of Image Processing applications, each multi-bit element (pixel) of the 2D data set (image) comes in parallel from the acquisition sensor (camera, VCR, scanner,...) following the scheme of figure 1.a. Each element, in fact, carries the information of a single pixel, and this second solution would need a hardware *data shuffler* for data transposition. The Connection Machine CM-2 [7] implements a hardware extension (*Sprint-chip*) explicitly devoted to this purpose.

The first prototype of the PAPRICA system implements the solution depicted in figure 1.a, but in the next version [3] the external memory will be organized as in figure 1.b, and a proposal of a specific hardware data-shuffler with real-time performances is currently under evaluation.

This work presents a theoretical analysis leading to the determination of the hardware efficiency in the case of systems implementing the first solution (figure 1.a). The main result of this analysis lays in the determination of a technique (formalized in an automated procedure for the PAPRICA system) aimed to increment the hardware efficiency. This algorithm will

- (a) reorganize the data contained into the external memory;
- (b) perform the processing exploiting the hardware features in a highly efficient way; and
- (c) restore the result in the original format into the external memory.

Next section presents the specific virtualization mechanism implemented when a low amount of memory is owned by each PE (the *External Virtualization* mechanism), determining the parameters on which the processing speed depends. Section 3 provides a technique aimed to increment the processing speed by exploiting the bus parallelism. Section 4 presents the implementation of this technique on the PAPRICA system and shows the speed up in the case of a thinning filter iterated on binary images. Finally section 5 ends the paper with some concluding remarks.

2 The External Virtualization Mechanism

When the number of PEs is smaller than the number of elements in the data set, as generally happens

in Image Processing applications, a mechanism must be provided to virtualize the PA. The *External Virtualization* mechanism [2] is a widely used [11, 8, 9, 5, 6] technique for PE virtualization, generally implemented on low-cost SIMD architectures whose simple PEs only own a small amount of memory. The computation is serialized in windows: the PA is loaded from the external memory with a sub-window of the data set; then the computation is performed until a special instruction (hereinafter referred to as **UPDATE** to reflect the one used in the PAPRICA system) is reached; and finally the results are stored back again into the external memory. These steps are iterated until all the sub-windows have been processed. Then the first sub-window is reloaded again into the PA and the computation is resumed until the next **UPDATE** operation is found.

The main problem of this kind of virtualization mechanism is due to the fact that when the PA is loaded with a sub-window of the data set, the PA border processors cannot access their complete neighborhood. In fact, after the execution of a single instruction involving a 3×3 neighborhood, the values stored in the PA border processors are meaningless. If more 3×3 neighborhood-based operations are executed, then also the PEs whose neighbors contain meaningless data cannot produce valid results. Thus only a portion (hereinafter referred to as the *Validity Area*, VA) of the data processed by the PA contains significant data. The next windows that will be loaded into the PA will be partially overlapped with the previous ones, in order to correctly evaluate the results which in the previous computation were invalid.

2.1 Performance analysis

This section presents the performance analysis of a generic SIMD system, following the scheme shown in figure 2.

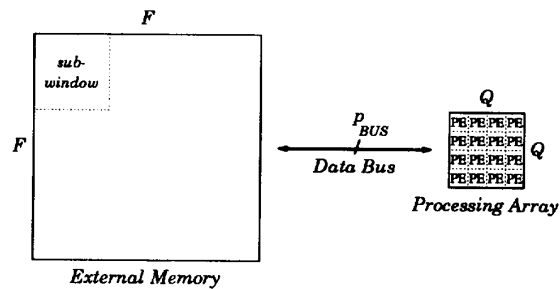


Figure 2: A system implementing the External Virtualization Mechanism.

Let's assume a system composed of a $Q \times Q$ PA connected to the external memory (logically organized as in figure 1.a) through a bus with parallelism p_{BUS} ; the basic transfer operation duration is T_M , while the PE instruction duration is T_C .

Let's assume that the data set to be processed by the system is a 2-dimensional array ($F \times F$) of elements with word parallelism¹ w . Parameters p_L and p_S represent the number of times the bus is used to transfer a single element of the data set, and are defined as the smallest integers greater than the ratio between the data set word parallelism and p_{BUS} during loading and storing operations respectively:

$$p_L = \left\lceil \frac{w_L}{p_{BUS}} \right\rceil, \quad \text{and} \quad p_S = \left\lceil \frac{w_S}{p_{BUS}} \right\rceil, \quad (1)$$

where subscripts L and S stand for *loading* and *storing*.

The generic application that is executed by the system is composed of L instructions, G of them being *graphic operations*, namely involving the access to the PE's 3×3 neighborhood. The remaining $L - G$ operations are *logical operations* performed using as input only the data stored in each PE internal memory.

As a consequence of the application of graphic operations, as already shown in section 2, the VA is reduced. In order to avoid that the VA is completely shrunk, the sequence of program instructions is interleaved with n_{upd} UPDATE operations. Their presence splits the program into segments (here defined **UPDATE blocks**) which are executed in sequence on the whole data set, as they were independent programs. The determination of the number and position of UPDATE operations may influence the performance (in terms of processing speed) in a deeper way than the choice of the processing algorithm itself. The determination of an automated procedure for the optimal position of UPDATE operations have been analyzed in [2], where the determination of the processing speed (as a function of the above introduced parameters) is also presented.

According to [2], the processing speed S_{pr} is given by:

$$S_{pr} = \frac{\left(Q - 2 \frac{G}{n_{upd}}\right)^2}{(\overline{p_L} + \overline{p_S}) Q^2 T_M n_{upd} + L T_C}. \quad (2)$$

Note that p_L and p_S depend on the specific elaboration performed by each UPDATE block. In equation (2) p_L

¹In Image Processing applications the data set word parallelism represents the number of bits per pixel.

and p_S have been substituted by $\overline{p_L}$ and $\overline{p_S}$ to indicate their average values within a given application.

The processing speed is in fact a function of parameters that depend on the system technology and architecture (T_M, T_C, Q), on the specific computational task (L, G), on the specific coding² of the data set ($\overline{p_L}, \overline{p_S}$), and on the number of UPDATE instructions (n_{upd}).

According to the theoretical analysis developed in [2] for the determination of the optimal value for n_{upd} (as a function of the architectural and application-dependent parameters), it follows that

$$\left[n_{upd}\right]_{opt} = 6 \frac{G}{Q}. \quad (3)$$

Combining equations (2) and (3), the processing speed in the case of optimal positioning of UPDATE operations can be expressed by:

$$S_{pr} = \frac{4}{9} \frac{Q^2}{6 (\overline{p_L} + \overline{p_S}) Q G T_M + L T_C}. \quad (4)$$

Assuming $T_M \simeq T_C$, as far as L is negligible with respect to $6 (\overline{p_L} + \overline{p_S}) Q G$ (for example when Q is sufficiently large), the dependence of the processing speed on the length of the program is negligible.

Since the system parameters (Q, T_M, T_C) are fixed by the physical architecture, and factors G and L are determined by the specific application, the processing speed can be increased by determining an efficient data coding: the tuning of parameters $\overline{p_L}$ and $\overline{p_S}$ for the optimization of the processing speed is the topic of the next section.

3 Incrementing the hardware efficiency

This section provides a way to improve the system performance (the processing speed) based on the maximization of the hardware efficiency. With the maximization of the PA efficiency it has been possible [2] to obtain equation (3) for the optimization of the number and position of UPDATE operations. Let's now focus on the maximization of the data bus efficiency.

The efficiency of the data bus η during transfer operations is defined as the ratio between the word parallelism of the data to be transferred and the number of bits actually transferred (which is given by the number of times the bus is used to transfer a single element of

²For example a *compressed* format vs a *plain* format of the input and temporary data sets.

the data set times the data bus parallelism):

$$\eta \triangleq \frac{\text{data_set_parallelism}}{\text{number_of_transfers} \times \text{bus_parallelism}} \leq 1. \quad (5)$$

As in the case above, let's distinguish between data bus efficiency during loading and storing operations:

$$\eta_L \triangleq \frac{w_L}{p_L p_{BUS}} \quad \text{and} \quad \eta_S \triangleq \frac{w_S}{p_S p_{BUS}}. \quad (6)$$

It is easy to deduce that, according to equations (1) and (6),

$$\eta_L \begin{cases} = 1 & \text{if } \frac{w_L}{p_{BUS}} \text{ is an integer} \\ < 1 & \text{if } \frac{w_L}{p_{BUS}} \text{ is not an integer} \end{cases} \quad (7)$$

and

$$\eta_S \begin{cases} = 1 & \text{if } \frac{w_S}{p_{BUS}} \text{ is an integer} \\ < 1 & \text{if } \frac{w_S}{p_{BUS}} \text{ is not an integer} \end{cases}. \quad (8)$$

The maximization of the data bus efficiency is then reduced to the use of a data set word parallelism in transfer operations that is a multiple of the data bus parallelism p_{BUS} . For sake of simplicity, from now on η will be used instead of η_L and η_S , and w instead of w_L and w_S to parametrize a generic data transfer between the PA and the external memory.

Considering as an example the PAPRICA system, where $p_{BUS} = 16$, when an UPDATE block operates on a binary image, the efficiency of the data bus η is considerably low. In fact η becomes:

$$[\eta]_{\text{binary_images}} = \frac{1}{16} \simeq 6\%. \quad (9)$$

As shown in equations (8) the data bus efficiency η can be incremented if all the data flowing through the bus (p_{BUS} bits each memory cycle) would held significant data. This can be achieved if the data transfer process is preceded by a data packing procedure, which stacks more elements of the original data set onto a single element of a new packed data set. The new ideal data set (still a 2D array) will contain a different number of elements with respect to the original one, and each element will have a word parallelism which is a multiple of p_{BUS} . Note that the new data set is obtained through a packing procedure, and not through a data compression procedure: data are only grouped and the data coding is not modified.

3.1 Data reorganization and packing

In order to simplify the comprehension of the following analysis, the processing of images is considered. Images are 2D pixel sets, where w indicates the image depth in bit/pixel.

Let us consider two constants $q_x, q_y \in N^+$ such that $q_x \times q_y \times w$ is as close as possible (but smaller or equal) to a multiple of p_{BUS} . The portion of the image memory which contains the $F_x \times F_y \times w$ processing frame can be rearranged and remapped onto another image memory area with dimensions $\left(\frac{F_x}{q_x}\right) \times \left(\frac{F_y}{q_y}\right) \times (q_x q_y w)$. This means that the number of sub-windows in which the external virtualization mechanism splits the image is sensibly reduced, but the major benefit lays in the reduction of factor G . In fact, as shown in figure 3, through a single 3×3 neighborhood-based operation each PE has access to the information stored in a wider *logical neighborhood*³, whose dimensions depend on the pixel position within the $q_x \times q_y$ sub-window. The *maximum logical neighborhood* accessible by every pixel with a single 3×3 neighborhood-based operation corresponds to a square of $(2 q_x + 1, 2 q_y + 1)$. The execution of G operations based on a 3×3 neighborhood on a packed image means that a $(2 G q_x + 1, 2 G q_y + 1)$ *logical neighborhood* is accessed.

Thus, assuming $q_x = q_y = q$, the relation between factor G when processing a plain or a packed image is:

$$[G]_{\text{packed}} = \frac{[G]_{\text{plain}}}{q}. \quad (10)$$

The same considerations apply to the number of

UPDATE instructions $[n_{upd}]_{\text{packed}} = \frac{[n_{upd}]_{\text{plain}}}{q}$, and

to the linear frame dimension $[F]_{\text{packed}} = \frac{[F]_{\text{plain}}}{q}$.

Moreover, since a q^2 image sub-window is associated to a single PE, each PE must repeat q^2 times the same elaboration (a few considerations and some optimizations that could shorten the final program length are presented in section 4.2), leading to the following relation:

$$[L]_{\text{packed}} = q^2 [L]_{\text{plain}} \quad (11)$$

Assuming $p_L = p_S = 1$ for sake of simplicity, the

³Note that the *logical neighborhood* is different from the PE physical neighborhood which is fixed by the hardware architecture: the *logical neighborhood* refers to the data set neighborhood that can be accessed within a single operation.

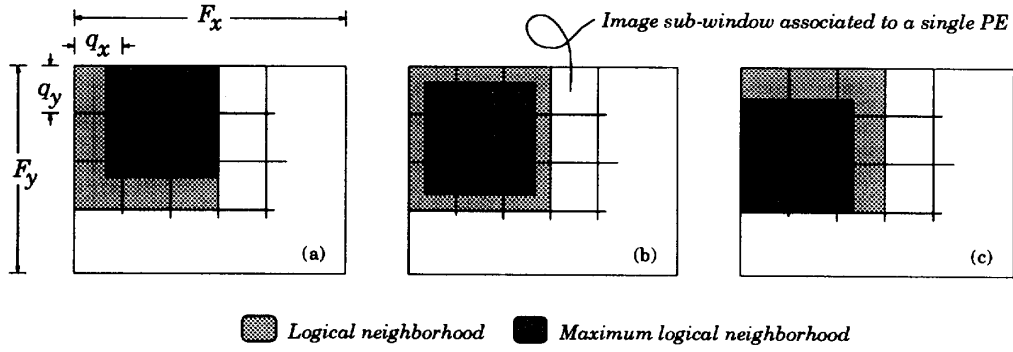


Figure 3: A pixel in a packed image: the dimensions of its *logical neighborhood*, reachable through a single 3×3 neighborhood-based operation, depend on the pixel position within the $q_x \times q_y$ sub-window. Conversely, the *maximum logical neighborhood* is independent of the pixel position.

processing speed is then expressed by

$$[S_{pr}]_{packed} = \frac{4}{9} \frac{Q^2}{\frac{12 Q G T_M}{q} + q^2 L T_C}, \quad (12)$$

instead of

$$[S_{pr}]_{plain} = \frac{4}{9} \frac{Q^2}{12 Q G T_M + L T_C}, \quad (13)$$

where n_{upd} , G , and L are referred to the application working on a plain image. Now, for sufficiently large values of q , $q^2 L T_C$ is no more negligible with respect to $\frac{12 Q G T_M}{q}$ in equation (12). The most important consideration is anyway due to the fact that when processing a packed image, the frame dimensions are reduced. The processing time, defined as

$$T_{pr} \triangleq \frac{(F)^2}{S_{pr}}, \quad (14)$$

is in fact given by:

$$\begin{aligned} [T_{pr}]_{packed} &= \frac{\left(\frac{F}{q}\right)^2}{[S_{pr}]_{packed}} = \\ &= \frac{9}{4} \frac{F^2}{Q^2} \left(\frac{12 Q G T_M}{q^3} + L T_C \right) \end{aligned} \quad (15)$$

instead of

$$[T_{pr}]_{plain} = \frac{F^2}{[S_{pr}]_{plain}} = \frac{9}{4} \frac{F^2}{Q^2} (12 Q G T_M + L T_C). \quad (16)$$

Equations (15) and (16) show that the higher q , the higher the benefit of processing a packed data set instead of a plain one. Anyway, it must be highlighted that the total time T_{tot} required for the processing

- matches $[T_{pr}]_{plain}$ in the case of a plain image, but
- is larger than $[T_{pr}]_{packed}$ in the case of a packed image, since two data reorganization procedures must be performed:

$$[T_{tot}]_{plain} = [T_{pr}]_{plain}, \quad (17)$$

while

$$[T_{tot}]_{packed} = T_{packing} + [T_{pr}]_{packed} + T_{unpacking}. \quad (18)$$

4 Data reorganization on **PAPRICA** system

In the case of PAPRICA architecture (where $p_{BUS} = 16$), when processing binary images ($F \times F \times 1$) the optimal choice is $q_x = q_y = 4$. The *maximum logical neighborhood* accessed when $G = 1$ is therefore 9×9 . Table 1 shows the dimensions of the *maximum logical neighborhood* accessed in the case of a plain image or a packed one.

Figure 4 shows how the reorganization of a binary image takes place on PAPRICA system: a $Q \times Q \times p_{BUS}$ sub-window is loaded from the external memory (area A) to the PA with a data bus efficiency $\eta_L = \frac{1}{p_{BUS}}$; then data are reorganized and moved within the

Reduction Factor	Logical neighborhood (plain image)	Logical neighborhood (packed image)
$G = 0$	1×1	1×1
$G = 1$	3×3	9×9
$G = 2$	5×5	17×17
$G = 3$	7×7	25×25
$G = 4$	9×9	33×33

Table 1: The *maximum logical neighborhood* in the case of a plain and packed image for $q = 4$.

PE internal memories; finally a $\frac{Q}{q} \times \frac{Q}{q} \times p_{\text{BUS}}$ subset of the PEs data is stored into the image memory (area B) with a higher data bus efficiency $\eta_S = \frac{q^2}{p_{\text{BUS}}}$.

This behavior is possible because PAPRICA system allows different mappings between the PEs internal memories and the external memory during load and store procedures. Moreover, it is possible to set a **MARGIN** parameter to $\frac{q-1}{2} Q$, so that only the central square array $\left(\frac{Q}{q} \times \frac{Q}{q}\right)$ of the PA values is saved back into the image memory.

The time needed to reorganize a $F \times F \times 1$ image, with $q = 4$, is given by the sum of:

- (a) the loading time $F^2 T_M$;
- (b) the reorganization time $\left(\frac{F}{Q}\right)^2 L_r T_C$; and
- (c) the saving time $\left(\frac{F}{4}\right)^2 T_M$:

$$T_{\text{packing}} = F^2 T_M + \left(\frac{F}{Q}\right)^2 L_r T_C + \left(\frac{F}{4}\right)^2 T_M \simeq \simeq F^2 \left(T_M + \frac{L_r T_C}{Q}\right), \quad (19)$$

where L_r represents the number of Assembly instructions needed by the data reorganization routine.

Similar considerations apply to the data unpacking procedure, which requires the same amount of time to restore the resulting packed data set into a plain one:

$$T_{\text{unpacking}} = T_{\text{packing}}. \quad (20)$$

4.1 Performance analysis on system

As an example, let us consider a single thinning [10] iteration, which in [1] is reduced to a sequence of 8

matchings with 3×3 patterns $\left(\left[G\right]_{\text{plain}} = 8\right)$, corresponding to a 17×17 *image neighborhood*. Assuming $Q = 16$, it corresponds to a program formed by 3 **UPDATE** blocks. After the data packing procedure (with $q = 4$), the VA reduction factor becomes $\left[G\right]_{\text{packed}} = 2$, and thus $\left[n_{\text{upd}}\right]_{\text{packed}} = 1$. Considering that $\left[L_r\right]_{\text{plain}} \simeq 50$ and $\left[G\right]_{\text{plain}} = 8$, the time required to filter a 256×256 image with a single thinning iteration on PAPRICA system ($Q^2 = 256$, $T_M = 350$ ns, $T_C = 350$ ns) is $\left[T_{1\text{-iter}}\right]_{\text{plain}} = 237$ ms, while $\left[T_{1\text{-iter}}\right]_{\text{packed}} = 13$ ms. The sum of the time T_{packing} and $T_{\text{unpacking}}$ required for the two $1 \rightarrow q^2$ and $q^2 \rightarrow 1$ data reorganization procedures is about 2×36 ms = 72 ms, thus even the execution of a single iteration is sufficient to justify the data reorganization⁴.

Table 2 shows the execution timings of a different number of iterations of a thinning filter on a plain image $\left(\left[T_{\text{tot}}\right]_{\text{plain}}\right)$, on a packed image $\left(\left[T_{\text{tot}}\right]_{\text{packed}}\right)$, and the relative speed-up percentage, defined as:

$$\text{Speed-up} \triangleq 100 \left[\frac{\left[T_{\text{tot}}\right]_{\text{plain}}}{\left[T_{\text{tot}}\right]_{\text{packed}}} - 1 \right]. \quad (21)$$

Iterations	$\left[T_{\text{tot}}\right]_{\text{plain}}$	$\left[T_{\text{tot}}\right]_{\text{packed}}$	Speedup %
# 1	237 ms	85 ms	180 %
# 2	475 ms	98 ms	380 %
# 4	712 ms	111 ms	540 %
# 8	950 ms	124 ms	660 %

Table 2: Performance of a thinning filter working on a 256×256 plain and a 64×64 packed version of the same image.

In order to exploit this optimization criterion, a program has been developed for the automatic translation of a generic PAPRICA Assembly code working on binary data sets to its equivalent version working on packed data.

⁴A new data packing technique is currently under study and evaluation, based on a more efficient exploitation of PAPRICA hardware features: it should allow to decrease the data reorganization time to about 18 ms.

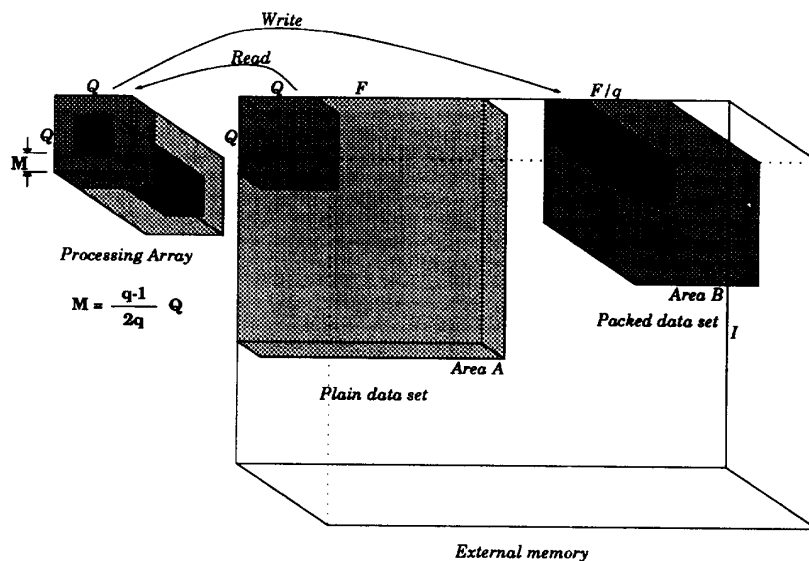


Figure 4: The data reorganization procedure on PAPRICA system.

4.2 The automatic code conversion

This section presents an automatic procedure used to convert a PAPRICA Assembly program working on binary images into its equivalent version working on 16 bit packed images ($q = 4$).

According to equation (11), each single-bit instruction will be converted into a sequence of q^2 instructions. The following example shows how a single instruction is converted. Assuming that layer 0 in the

- | | |
|-------------|-----------------------|
| L16 = (L4) | L24 = (L12) |
| L17 = (L5) | L25 = (L13) |
| L18 = (L6) | L26 = (L14) |
| L19 = (L7) | L27 = (L15) |
| L20 = (L8) | L28 = NMOV(L0) |
| L21 = (L9) | L29 = NMOV(L1) |
| L22 = (L10) | L30 = NMOV(L2) |
| L23 = (L11) | L31 = NMOV(L3) |

Note that, since each PE memory contains a 4×4 logical neighborhood, the previous sequence is formed by 12 mere data transfers within the PE memory and 4 read operations from the PE neighborhood.

Without entering too much into the detail of the implementation, only the two following optimization criteria will be highlighted.

- (a) Due to the specific implementation of PAPRICA instruction set, some complex instructions (such as the 8-direction dilation **EXP**) must be considered as a combination of 8 translations, and thus their equivalent version working on packed sets will be formed by 8×16 instructions, instead of only 16.
- (b) On the other hand, instead of converting one instruction at a time, in some cases it is possible to convert a set of instructions, generating shorter programs working on packed sets.

As shown in equation (12), for sufficiently large values of q , the length of the program is no more negligible when processing packed sets, and thus the role of

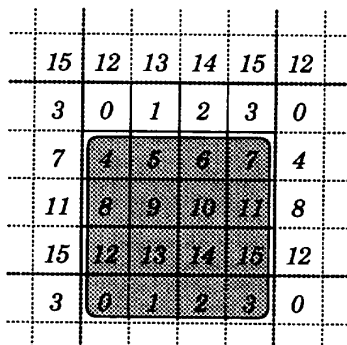


Figure 5: A north translation in a packed image

plain image maps onto layers $0 \div 15$ in the packed image, layer 1 onto layers $16 \div 31$ and so on, instruction L1 = **NMOV(L0)** (binary layer 1 is loaded with the north translation of layer 0) will be converted, according to figure 5, into the following sequence:

the code optimizer is of basic importance. Case (a) highlights that it is not worth to optimize the original code working on plain sets, since the optimizations cannot be transferred to the converted code. On the other hand, case (b) shows that sometimes it is possible to convert some specific code configurations into programs which are shorter than what would be obtained with an instruction-wise translation. The advantages highlighted by case (b) depend on the specific code segment to be converted, but it has been shown that it is possible to achieve a 60 % reduction in the number of instructions of the output code.

5 Conclusions

Starting from the analysis of the performance of SIMD systems implementing the External PE Virtualization Mechanism, a critical point in the design of the system hardware has been addressed. In particular it has been proven how the external memory logical organization together with the data bus parallelism play fundamental roles in the determination of the system performances.

This paper presented a technique for performance improvement on the PAPRICA system, based on data packing. An example (thinning of binary images) has shown that even if this technique requires pre- and post-processings to reorganize the data sets, the total time needed to pack, process, and unpack the data is remarkably shorter than the time needed by the original algorithm working on a plain data set.

An optimizing Assembly-to-Assembly translator has been developed for the automatic conversion of a generic PAPRICA Assembly program working on binary data sets into its equivalent version working on packed data sets.

Acknowledgments

The author would like to thank Filippo Labadini and Alessandro Bonani for their help in the development and testing of PAPRICA Assembly code.

References

- [1] C. Arcelli, L. Cordella, and S. Levialdi. Parallel Thinning of Binary Pictures. *Electronics Letters*, 11:148-149, July 1975.
- [2] A. Broggi. Performance Optimization on Low-Cost Cellular Array Processors. In *Proceedings MPSCS - IEEE International Conference on Massively Parallel Computing Systems*, Ischia, Italy,

- May 2-6 1994. IEEE Computer Society - EuroMicro. In press.
- [3] A. Broggi, G. Conte, G. Burzio, L. Lavagno, F. Gregoretti, C. Sansoè, and L. M. Reyneri. PAPRICA-3: A Real-Time Morphological Image Processor. In *Proceedings ICIP - First IEEE International Conference on Image Processing*, Austin, TX, November 13-16 1994. IEEE Computer Society. In press.
- [4] A. Broggi, G. Conte, F. Gregoretti, C. Sansoè, and L. M. Reyneri. The PAPRICA Massively Parallel Processor. In *Proceedings MPSCS - IEEE International Conference on Massively Parallel Computing Systems*, Ischia, Italy, May 2-6 1994. IEEE Computer Society - EuroMicro. In press. Awarded by IEEE Computer Society as Best Paper presenting a European prototype of a massively parallel system.
- [5] G. Conte, F. Gregoretti, L. Reyneri, and C. Sansoè. PAPRICA: a Parallel Architecture for VLSI CAD. In A.P.Amblar, P.Agrawal, and W.R.Moore, editors, *CAD Accelerators*, pages 177-189. North Holland, Amsterdam, 1991.
- [6] R. Cucchiara, M. Piccardi, L. D. Stefano, M. Monacelli, G. Rustichelli, and T. S. Cinotti. The Prototype of a Heterogeneous Machine for Robot Vision: Architecture, Programming and Performance. In *PSF94 - Parallel System Fair 94 - IEEE International Symposium of Parallel and Distributed Systems*, Cancun, Mexico, April 1994.
- [7] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, Ma., 1985.
- [8] NCR Corporation, Dayton, Ohio. *Geometric Arithmetic Parallel Processor*, 1984.
- [9] S. Reddaway. DAP - A distributed array processor. In *1st Annual Symposium on Computer Architectures*, pages 61-65, Florida, 1973.
- [10] A. Rosenfeld. A characterization of parallel thinning algorithms. *Information Control*, 29:286-291, 1975.
- [11] 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.