Systolic array for computing the pixel purity index (PPI) algorithm on hyper spectral images

Dominique Lavenier, Erwan Fabiani, Steven Derrien, Charles Wagner
IRISA, Campus de Beaulieu, 35042 Rennes cedex,
France
lavenier@irisa.fr

ABSTRACT

The Pixel Purity Index (PPI) algorithm is used as a pre-processing to find end-members in a hyper spectral image. It tries to identify pure spectra by assigning a pixel purity index to each pixel in the image. The algorithm proceeds by generating a large number of random vectors through the hyper spectral image and by computing a dot-product between each vector and all the pixels. Since the number of random vectors is high (a few thousands), this algorithm may require hours of computation on standard computers.

We present a systolic implementation of the PPI algorithm. It is based on a linear systolic array connected to a host processor through its external I/O bus system. In this scheme, the image is stored on the host processor memory and flushed several times through the array. The performance is mainly dictated by the I/O bus bandwidth and the ability to implement large systolic arrays: the fewer the passes needed through the array, the better the performance.

The hardware implementation targets Xilinx Virtex boards, but the specification is independent of the platform: no external memories are required and the architecture works whatever the size of the linear systolic array. Experiments carried out on a low-cost reconfigurable board (a single Xilinx Virtex 800) show a speed-up of two orders of magnitude compared to a software implementation.

Keywords: Hyper spectral, Dot-Product, Pixel Purity Index, FPGA, Reconfigurable, Systolic Architectures. Place-and-Route.

1. INTRODUCTION

A hyper spectral image is a 2-dimensionnal array of hyper pixels. These elements can be viewed as vectors of *D* data, each of them representing a specific value for a given wave length among hundreds of spectral bands. An assumption is that an hyper spectral image contains relatively few materials and the hyper pixels (called simply pixel in the following) are a mixture of *pure* materials. The aim of the Pixel Purity Index (PPI) algorithm is to help identifying the pure pixels, so that all the remaining pixels can be expressed as a linear combination of them.

The goal of the paper is not to explain how the PPI algorithm works. Readers interested in the PPI foundation can refer to [1] and [2] for a complete explanation. We will just briefly recall its principle. The PPI algorithm proceeds by generating a large number of random *D*-dimensional vectors, called skewers, through the hyper spectral image as shown in figure 1. For each skewer, every data point is projected onto the skewer, and the position along the skewer is noted. The data points which correspond to extrema in the direction of a skewer are identified, and placed into a list. As more skewers are generated, this list grows. The number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the purest, and a pixel's count provides its pixel purity index.

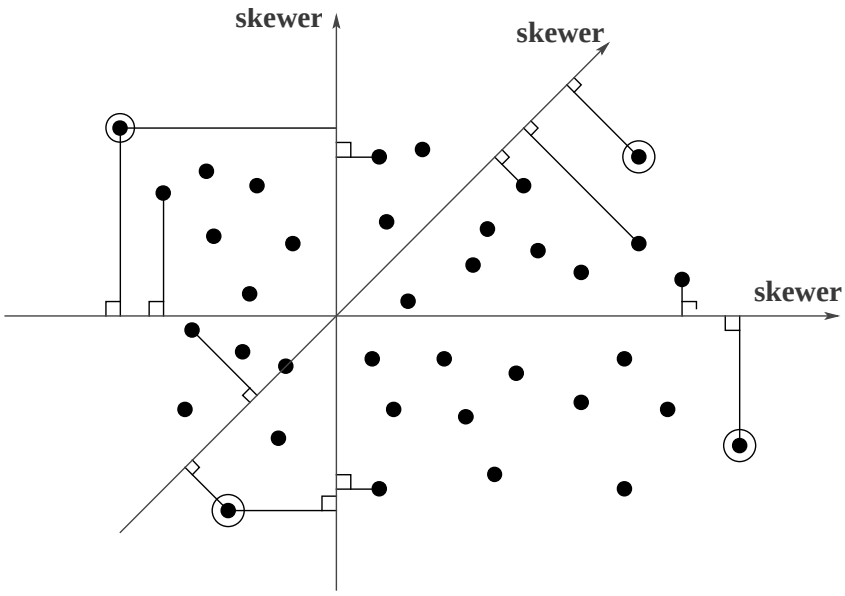


Figure 1. The PPI algorithm works by projecting points in the data set (black points) onto random skewers (arrows). For each skewers, 2 extreme points are identified, and their pixel purity index is incremented. In the figure above, the circled points are identified as candidate because their projection onto one or more skewers is extreme.

In this algorithm, the most time consuming part is the calculation of the projections. To detect pure pixels, thousands of skewers must be generated, and a projection (actually a dot-product) with all the pixels of the image is performed. The complexity *C* of the PPI algorithm, in terms of elementary MAC (Multiplication/Accumulation) operation, is:

$$C = N \times K \times D$$

N is the number of pixels, *K* the number of skewers, and *D* is the number of spectral bands. For example, processing the PPI algorithm on a hyper spectral satellite image of 512×614 pixels of 224 bands with 10 000 skewers requires the calculation of more than 7×10^{11} MACs (Multiplication/Accumulation), that is a few hours of non-stop computation with a 500 MHz microprocessor.

Fortunately, the PPI algorithm is well suited for parallelism. The computation of all the projections are independent and can be performed simultaneously, leading to many ways of parallelization. In [3], two parallel architectures are proposed. Both are based on a 2-D processor array tightly connected to a few memory banks. A speed-up of 80 is obtained through an FPGA implementation on the *Wildforce* board (4 Xilinx XC4036EX + 4 memory banks of 512 Kbyte) [7]. Although this work has demonstrated the efficiency of a hardware implementation on a reconfigurable board, the solution is not scalable. As a matter of fact, the design is tailored to the *Wildforce* board and it cannot be reused without huge modifications for another board.

The architecture we present in this paper aims to overcome this drawback. Again, reconfigurable boards are targeted but the architecture specification is independent of the platform: no external memories are required and the architecture is scalable depending of the amount of available resources.

The rest of the paper is organized as follows: section 2 explains how a systolic array is derived from the PPI algorithm. Section 3 focuses on the processor architecture. Section 4 estimates the performances of the systolic architecture and highlight the limitations when connecting a reconfigurable board through the IO bus. Section 5 is an experimentation carried on a low-cost reconfigurable board (the Spyder board) where a speed-up of 120 is achieved compared to a software version running on a 500 MHz microprocessor. Section 6 concludes this paper.

2. SYSTOLIC ARRAY

As mentioned before, the PPI algorithm consists of computing a very large number of dot-products, and all these dot-products can be performed simultaneously. A possible way of parallelization is to have a system able to compute *K* dot-products in the same time against the same pixel (*K* is the number of skewers). Supposing such a system, the PPI algorithm can be written as follows:

```
for (n=0; n<N; n++)
  {
   forall (k=0; k<K; k++)
     {
      dp[k] = dot-product (pixel[n],skewer[k]);
      if (dp[k] < Min[k]) { Min[k]=dp[k]; iMIN[k]=n; }
      if (dp[k] > Max[k]) { Max[k]=dp[k]; iMAX[k]=n; }
      }
  }
```

The forall loop expresses that *K* dot-products are first performed in parallel, then *K* min and max are also computed in parallel. Now, if we suppose that we cannot simultaneously compute *K* dot-products but only a fraction *K/P*, then the algorithm is rewritten as:

```
for (p=0; p<P; p++)
  {
   x = p*(K/P);
   for (n=0; n<N; n++)
     {
      forall (k=0; k<K/P; k++)
       {
        dp[x+k] = dot-product (pixel[n],skewer[x+k]);
        if (dp[x+k] < Min[x+k]) { Min[x+k]=dp[x+k]; iMIN[x+k]=n; }
        if (dp[x+k] > Max[x+k]) { Max[x+k]=dp[x+k]; iMAX[x+k]=n; }
       }
     }
   }
```

The algorithm is then split into *P* passes, each pass performing $N \times K/P$ dot-products. From an architectural point of view, we can translate the forall loop into an array of *K/P* processors. Each processor receives successively the *N* pixels, computes *N* dot-products, and kept in memory the two pixels having produced the min and the max dot-products. In this scheme, each processor holds a different skewer which must be input before each new pass. Figure 2-A illustrates the principle. The skewer initialization is not shown.

As we hope to fit a large number of processors into the reconfigurable boards, broadcasting the pixels to every processor – maybe to a few hundred – is not recommended because of electrical signal fanout constraints. Getting back the results (iMIN and iMAX) is also an important issue which requires careful attention. A 1-D systolic array avoids broadcasting the pixel while simplifying the collection of the results. Figure 2-B represents this new architecture. Basically, a processor performs exactly the same computation as before. The only difference is that the processors work on different pixels. During the step 1 (systolic cycle 1) the processor 1 processes pixel 1. During the step 2, the processor 1 processes pixel 2 and the processor 2 processes pixel 1. And so on. After *K/P* systolic cycles all the processors are working. Since the number of pixels is high compared to the number of processors, the setup time provides negligible penalty. When all the pixels have been flushed through the array, *K/P* systolic cycles are thus required to collect the results. This is done by shifting the iMIN and iMAX indexes to the right of the array.

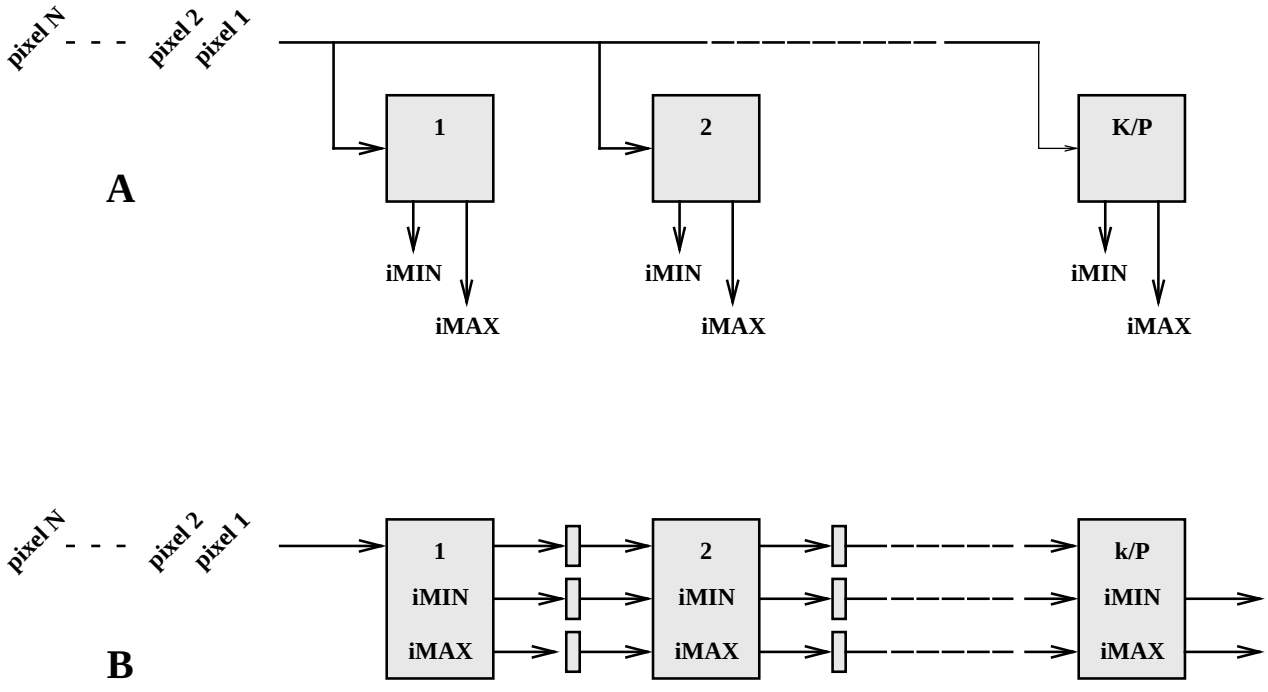


Figure 2. (A) principle of the parallelization: *K/P* processors perform successively *N* dot-products. The pixels are broadcast to all the processors. **(B)** The computation is pipelined (systolized) to avoid fanout problems and to provide a scalable system. Collecting the results is done by shifting to the right the iMIN and iMAX indexes once the N pixels have crossed the array.

The advantage of the systolic array is its scalability. Depending of the resources available on the reconfigurable board the number of processors can be adjusted without modifying the control of the array. Now, in order to reduce the number of passes, a maximum number of processors have to fit in the FPGA components. The next section describes the architecture of a processor according to these constraints.

3. PROCESSOR ARCHITECTURE

Basically, a systolic cycle consist in computing a dot-product between a pixel and a skewer, and to memorize the index of the pixel if the dot-product is higher or smaller than a previously computed max or min value. Remember that a pixel is a vector of *D* values, just like a skewer. A dot-product calculation (dp) is performed as follows:

$$dp = \sum_{i=1}^{D} pixel[i] \times skewer[i]$$

It requires *D* multiplications and *D*-1 additions. In [3] it has been shown that the skewer values can be limited to a very small set of integers when *D* is large, as in the case of hyper spectral images. A particular and interesting set is {1,-1} since it avoids the multiplication. The dot-product is thus reduced to an accumulation of positive and negative values.

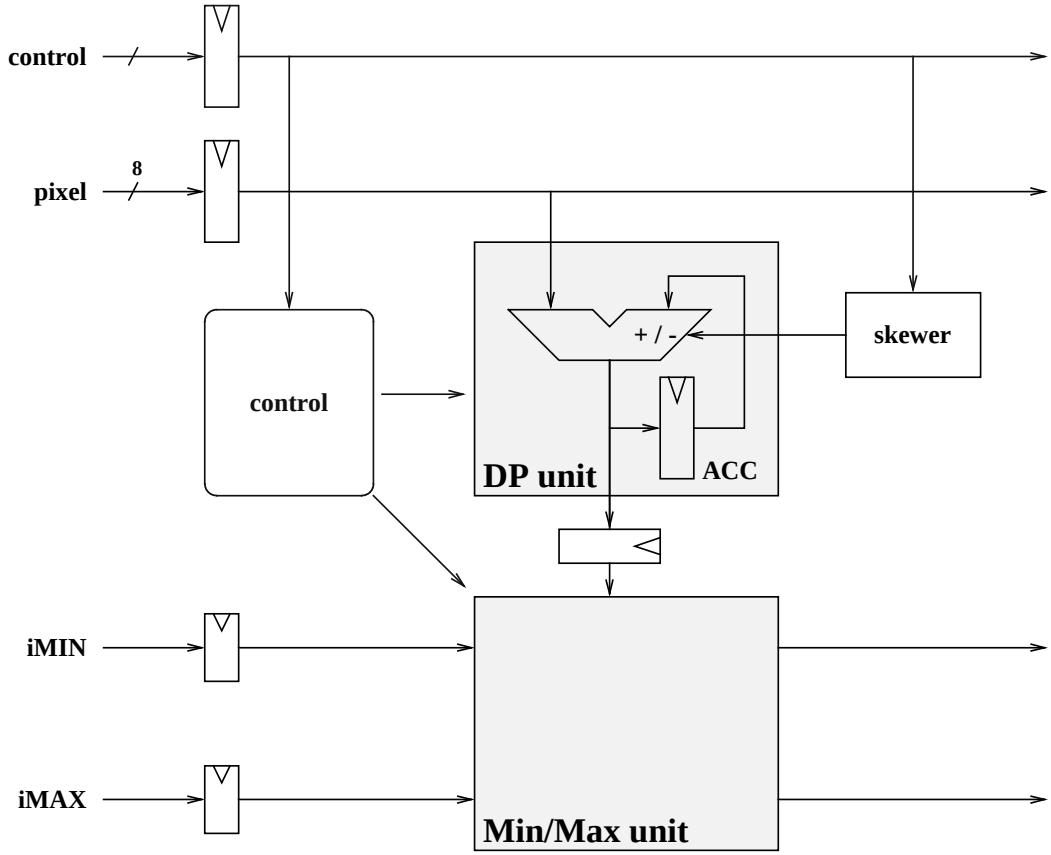
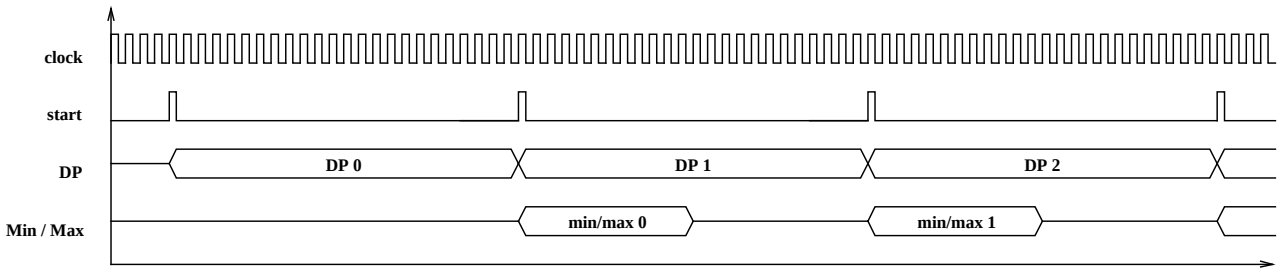


figure 3. Architecture of a processor: the DP unit computes the dot-product by summing up positive or negative pixel values. The Min/Max unit performs bit-serially a comparison between a min and a max value and store the pixel number if the dot-product exceed one of these two extreme values.

If we suppose that an addition or a subtraction is executed every clock cycle, then the calculation of a dot-product requires *D* clock cycles. The comparison with a max and a min value, and the index updating, is a quick operation compared to the dot-product. However, it requires a more complex hardware mechanism. In order to have the size of a processor as small as possible, this operation is done bit-serially, and in parallel with the computation of the next dot-product. The timing diagram below clarifies how a processor handles these different operations.



Let us suppose D = 24 (a multi spectral images split into 24 bands). A processor sequentially receives the 24 values of a pixel and accumulates them positively or negatively depending of the skewer components. This computation is done at the arrival speed of the pixel values. If we suppose that a new pixel value is available at each clock cycle, then 24 cycles are required as shown on the timing diagram where the start signal indicates the beginning of a new calculation. Before starting a new dot-product calculation, the result of the previous one is transmitted to a min/max unit which performs bit-serially the update of the min and max indexes. The constraint is that the number of clock cycles for processing the indexes must be smaller than D. This is the case since the number of spectra, here, is equals to 224 while about 32 clock cycles are needed to compute bit-serially the indexes.

Figure 3 depicts the architecture principle of a systolic processor. The **DP unit** accumulates the positive or negative values of the pixel input according to the skewer input. For instance, if the i^{th} component of the skewer is 0, then the i^{th} component of the pixel is summed up to the accumulator (ACC). If it equals 1, it is subtracted. This unit is only composed of a single 16-bit add/sub operator. The skewer is stored in a 1-bit D-word memory. The initialization mechanism is not represented.

The **Min/Max unit** receives a dot-product and has the charge of comparing this data to a min value and a max value. If it exceeds one of these two extrema, then the current dot-product value is substituted and the number of the pixel which has caused this change is memorized. Suppose, for example, that the (min, iMIN) and (max, iMAX) pairs contain respectively (-125,48) and (230,17). This means that until now, the lowest dot-product score has been produced by the pixel #48 and the highest dot-product score by the pixel #17. If the current dot-product is equal to –180, then the pair (-125,48) is replaced by the pair (-180,current pixel #). The min/max operation and the index updating are performed bit-serially and require 16 cycles for the comparison, plus $\log_2(N)$ cycles for updating. the index

4. PERFORMANCE ESTIMATION

The peak performance (P_{peak}) of the array is mainly determined by the dot-product capacity, that is the number of additions/subtractions executed in one second. It is expressed in millions of operations per second as:

$$P_{peak} = f \times K / P$$

f is the frequency in MHz and K/P represents the number of processors of the systolic array. The above formula supposes that the array is constantly feed: on each cycle a new data is available on its input. Unfortunately, this may not be the case, especially if we consider a reconfigurable board plugged trough the IO bus system of the micro-processor. The PPI algorithm proceeds into P passes, and each pass requires flushing the hyper spectral image from the main memory to the array. Thus, instead of considering that a data is present every clock cycle, it is better to consider the transfer capacity of the IO bus for estimating the average performance of the array.

Let us suppose a 8-bit encoded pixel component. Then the average performance (P_{avrg}) is given by:

$$P_{avrg} = BW \times K / P$$

BW is the bandwidth expressed in Mbytes/second. Now, if we want to estimate the time (T) for computing the PPI algorithm, the bandwidth is taken into consideration as follows:

$$T = P \times (N \times D) / BW$$

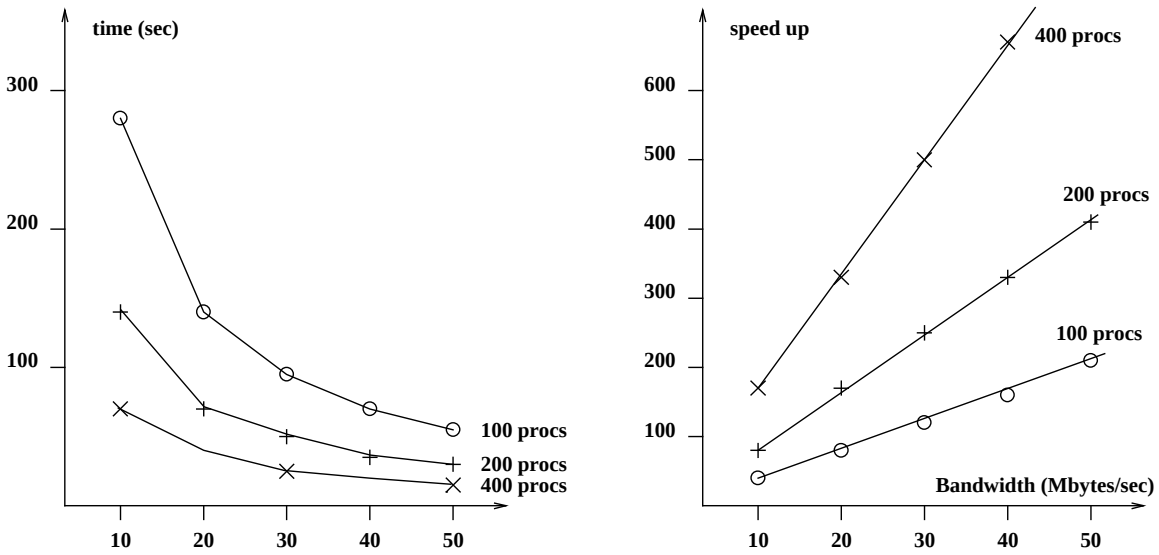


figure 4. Left: time, in second, for computing the PPI algorithm using a reconfigurable board connected to a PC through the IO bus. **Right:** speed-up compared to a software version running on a 500 MHz microprocessor. In both cases, the size of the hyper spectral image is 512 x 614 pixels of 224 spectral bands.

P is the number of passes, *N* the number of pixels and *D* the number of spectral bands. Figure 4 shows two diagrams: (1 – left side) the time for computing PPI on a 512 x 614 hyper spectral image of 224 spectral bands on a reconfigurable board connected to a microprocessor through its IO bus considering various bandwidths (from 10 Mbytes/sec to 50 Mbytes/sec) and for various lengths of arrays: 100, 200 and 400 processors; (2, right side) the speed-up compared to a 500 MHz microprocessor, again with a bandwidth ranging from 10 to 50 Mbytes/sec and a systolic array with 100 , 200 or 400 processors.

As can be seen, the speed-up can be very high, reducing hours of computation to a few tens of seconds. The next section validates these estimations on Xilinx Virtex components.

5. EXPERIMENTATION

A VHDL specification of the systolic array has been written and synthesized for different Xilinx Virtex components as shown below.

	XCV800	XCV1000 E	XCV 2000 E
# processors	224	288	448
% resources	88	86	85
Frequency (MHz)	40	57	50

A complete system (systolic array + PCI interface) has been implemented on a Spyder-Virtex- X2 / XCV-800 board from X2E [9]. This low-cost reconfigurable board houses a single Xilinx Virtex XCV 800 FPGA component and 2 Mbytes of memory (not used here). We measured an average PCI bandwidth of 15 Mbytes between the PC and the board, leading to a speed-up of 120 when running the PPI algorithm. In this experimentation, the performance is seriously limited by the transfer rate between the PC and the board: the array is able to absorb a pixel flow of 40 Mbytes/sec, while the PCI interface can only provide 15 Mbytes flow. This first experiment, however, demonstrates that a low-cost board (< $5000) and a non-optimized PCI connection (no DMA) can still yield good speed-up for the PPI algorithm.

We can extend this result to more recent (and more expensive) boards such as the Spyder-Virtex-X2E [10] from X2E or the *WildFire* board [11] from Annapolis micro systems, Inc., which can both house a Xilinx Virtex XCV1000-E or a XCV 2000-E FPGA component. With DMA transfer one can expect to reach very high speed-up: 400 with a Virtex 1000-E, and 600 with a Virtex 2000-E.

From an implementation point of view, in order to fit a maximum number of processors into the FPGA components, it is important to carefully manage the layout mapping of the systolic array. The locality of the processor interconnections have to be preserved to get the best frequency and to ensure that the FPGA can be routed. The systolic array has been placed-and-routed using **FRAP** (FPGA Regular Array Placer), a tool developed at IRISA for mapping regular array onto FPGA components [5]. Figure 5 details this environment: The input and output specification are written in the same formalism, respectively without and with placement directives. The placement is performed with the **FRAP** tool and acts in three steps [6]:

1. All possible shapes for a processing element are generated by combining all shapes of its sub-components.
2. A full *snake* placement is determined using the processing element shapes previously computed.
3. The final placement of the processing elements is performed according to their shapes.

The output of **FRAP** is an EDIF file, input to the vendor place-and-route tools. Steps 1 and 3 deal with processing element placement. We consider those elements rather small, i.e., a few operators essentially coming from a library, and that finding a good placement is a fast and non critical process. In step 2, the problem is to place a linear array on a bi-dimensional structure. The only way to keep two neighbor processing elements close to each other is to implement a snake-like arrangement of the array. The determination of the snake-like arrangement proceeds in two phases [6]: (1) divide the FPGA area in sub-areas and (2) for each area, place a maximum number of processing element in a snake-like fashion.

This placement strategy permits to optimize the use of the FPGA resources. For the PPI implementation the Virtex components are filled up to 85 %.

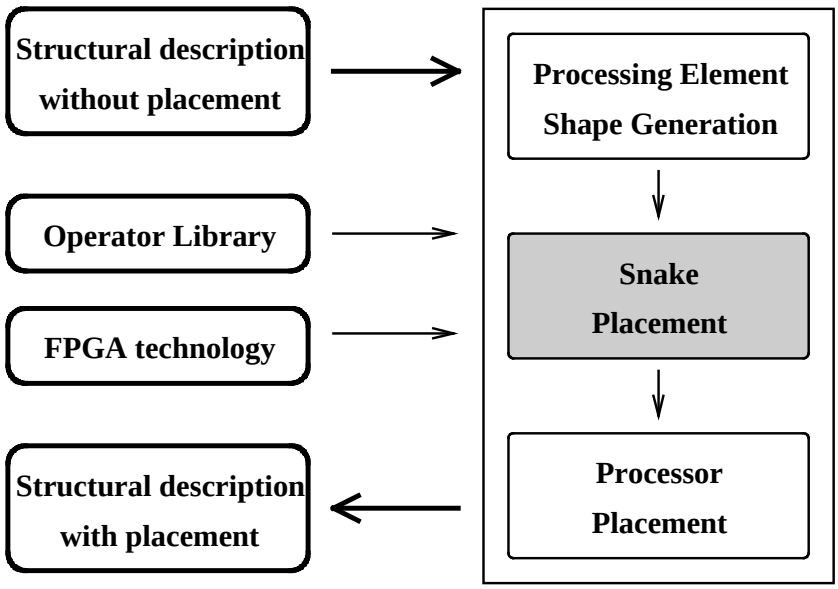


figure 5. FRAP: FPGA Regular Array Placer. From a structural description, the FRAP tools provides an equivalent structural description annotated with placement directives.

6. CONCLUSION

A systolic architecture for processing the PPI algorithm has been proposed and tested on a low-cost reconfigurable board. The originality of the processor architecture comes from its ability to handle parallel and bit-serial arithmetic, leading to a compact unit well suited for implementing large systolic arrays. The experimentation carried out on the Spyder board shows 2 orders of magnitude speed-up compared to a pure software version running on a 500 MHz microprocessor. Extrapolations to the next FPGA component generations indicate that a much higher speed-up is achievable since it directly depends on the number of processor on can fit inside a chip.

Housing and connecting a large number of processors into a single FPGA component is not straightforward. Good performance, both in terms of compactness and frequency, can only be achieved using high level placement tools. The methodology behind the FRAP environment has been successively tested for this application, allowing the designer to implement hundreds of small processors.

The major drawback of implementing the systolic architecture on a PCI-like board is that the transfer rate between the main microprocessor memory and the array seriously limits the overall performance. As estimated, a factor of 2 or 3 is lost if a standard and non-optimized protocol is used. DMA transfer is an alternative solution to increase the bandwidth. It requires a driver tuned to the board. Another possibility, is to adapt the systolic architecture to the bandwidth as proposed in [12] and [13]. The basic idea is to partition the array in such a way that a physical processor sequentially emulates several virtual processors. The virtual array is larger, but it works slower. For instance and ideally, instead of having 200 physical processors able to run at 50 MHz, but feed at a rate of 10 Mbytes, the virtual array can be extended to 1000 processors virtually running at 10 MHz. The reality is not so simple. The physical processors need extra memory storage, and the partitioning control for managing the array, sending the data and collecting the results is complex, but on the way to be automated.

REFERENCES

1. J.W. Boardman, “Automating spectral unmixing of AVIRIS data using convex geometric concepts”, *Summaries of the Fourth Annual JPL Airborne Geosciences Workshop*, R.Q. Green ed., 1994
2. J.W. Boardman, F.A. Kruse, R.Q. Greem, “Mapping target signature via partial unmixing of AVIRIS data,” *Summaries of the Fith Annual JPL Airborne Geosciences Workshop*, R.Q. Green ed., 1995
3. D. Lavenier, J. Theiler, J. Szymansky, M. Gokhale, J. Frigo, “FPGA Implementation of the Pixel Purity Index Algorithm,” *SPIE Photonics East, Workshop on Reconfigurable Architectures*, Boston, MA, 2000.
4. J. Theiler, D. Lavenier, N. Harvey, S. Perkins, J. Szymanski, “Using blocks of skewers for faster computation of pixel purity index,” *SPIE International Conference on Optical Science and Technology*, San Diego, CA, USA, 2000.
5. E. Fabiani, D. Lavenier, “Placement of Linear Arrays”, FPL 2000, 10th International Conference on Field Programmable Logic and Applications, Villach, Austria, 2000.
6. E. Fabiani and D. Lavenier. Using knapsack technique to place linear arrays on FPGA. Research report 1335, IRISA, June 2000.
7. Wildfore Reference Manual, technical report, revision 3.4, Annapolis Micro System Inc., 1999.
8. Virtex™ 2.5V Field Programmable Gate Arrays, Xilinx data sheet, DS003 (v.2.2), May 23, 2000.
9. Spyder-Virtex-X2 / XCV800, “Manual reference”, http://www.x2e.de, 1999.
10. Spyder-Virtex-X2E / XCV2000 E, “Manual reference”, http://www.x2e.de, 2001.
11. FireBird, Annapolis Micro Systems, Inc. http://www.annapmicro.com, 2000.
12. S. Derrien, S. Rajopadhye, “Loop Tiling for Reconfigurable Accelerators”, FPL 2001, International Conference on Filed Programmable Logic, Edimburg, 2001.
13. S. Derrien, S. Sur Kolay and S. Rajopadhye, “Optimal Partitionning for FPGA based Arrays Implementation”, IEEE PARELEC'0, Trois-Rivières, Quebec, 2000.