FPGA Implementation of the Pixel Purity Index Algorithm for Hyper-Spectral Images

LA-UR # 00-2466

March 2000

Dominique LAVENIER[1], James THEILER[2]

Los Alamos National Laboratory
[1]NIS-3: Space Data System
[2]NIS-2: Space and Remote Sensing Science
Los Alamos - NM 87545 - USA

Abstract

This technical report describes a hardware implementation of the Pixel Purity Index algorithm for hyper-spectral images. The realization has been experimented on a commercial FPGA board, the Wildforce board from Annapolis Micro System Inc. Speed-up of 80 has been achieved, reducing the computation time to several minutes compared to a few hours on standard sequential machines.

1 Introduction

The Pixel Purity Index (PPI) algorithm is part of a process for finding "end-members" in a hyper-spectral image, where the end-members can be defined as the spectral signature of this image. Basically a hyper-spectral pixel reflects the material composition of a given point of a particular scene, and the idea behind the notion of end-members is that a pixel is a linear combination of only a few distinct materials.

A geometrical interpretation of the linear combination in a D-dimensional Euclidean space is that the end-members are vertices of a convex polyhedron, and that the data are inside this polyhedron. The PPI algorithm attempts to identify the bounding D-dimensional polyhedron by finding vertices of the convex hull of data. These vertices correspond to particular data points.

The algorithm proceeds by generating a large number of random D-dimensional vectors, called skewers, through the hyper-spectral image. 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 on 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 most pure, and a pixel's count provides its "pixel purity index".

The complexity of the PPI algorithm is in $O(K \times D \times N)$ with K the number of skewers, D the

number of spectral bands and N the number of pixels. This is a very time consuming algorithm. As an example, processing a single 512×614 224-channel satellite image takes more than three hours on a 450 MHz processor (4K skewers). In that case, real-time analysis can only be achieved using specific architectures exploiting some interesting features of the PPI algorithm.

Fortunately, the PPI algorithm is a good candidate for hardware acceleration because it is readily parallelizable: all the computations perform on the K skewers are independent, leading to a high potential degree of parallelism, knowing that K is in the range of a few thousands!

The hardware implementation described in the next sections is heavily based of the ability to perform concurrently these computations onto FPGA components. The target reconfigurable system is the Wildforce accelerator board from Annapolis Micro System Inc [1], and the implementation we propose reduces the computation time to a few minutes compared to hours.

The next section presents the PPI algorithm and its parallelization. Section 3 gives the principle of the architecture and details the main components. Section 4 is devoted to the implementation on the Wildforce board.

2 Parallelization of the PPI Algorithm

The input data of the Pixel Purity Index algorithm are: (1) a hyper-spectral image composed of N pixels. Each pixel is a vector of D values of 16 bits; (2) a set of K random vectors (skewers) of size D. The output is an integer vector Q of size N.

As stated in the previous section, the key parameters of this algorithm are K, D and N which respectively represent the number of skewers, the number of spectral bands, and the number of pixels. The computation time essentially depends on these three parameters. The sequential C-like algorithm description is:

```
PIXELS[N][D];          // an image of N pixel vectors of size D
SKEWER[K][D];          // a set of K random vectors of size D (skewers)
Q[N];                  // the result array

for (n=0; n < N; n++) Q[n]=0;                // reset Q
for (k=0; k < K; k++)  {                     // for the K skewers
   for (n=0; n < N; n++) {                   // for the N pixels
      dp = 0;
      for (d=0; d < D; d++) {                // compute a Dot Product
         dp = dp + SKEWERS[k][d]*PIXELS[n][d];
       }
      if (dp > dpmax) { imax=n; dpmax=dp; }  // max computation
      if (dp < dpmin) { imin=n; dpmin=dp; }  // min computation
     }
   Q[imax]++;
   Q[imin]++;
}
```

For each skewer, N dot-products are computed in order to determine the two pixels which have

produced the higher and the lower dot-product. The Q vector is modified accordingly.

As stated before, the $N \times K$ dot-products can be performed independently. If we supposed that:

1. the hardware can support concurrently the computation of P dot-products,
2. NS pixels can be accessed simultaneously,
3. KS skewers can be accessed simultaneously

such that $P = NS \times KS$ then we can re-express the algorithm as follows:

```
for (k=0; k < K; k=k+KS) {
   for (n=0; n < N; n=n+NS) {
      forall (ks=0, ns=0; ks<KS, ns<NS; ks++, ns++) {
         dp[ks][ns] = Dot-Product(SKEWERS[k+ks],PIXELS[n+ns]);
         if (dp[ks][ns] > dpmax[k+ks]) {
            imax[k+ks]=n+ns; dpmax[k+ks]=dp[ks][ns];
          }
         if (dp[ks][ns] < dpmin[k+ks]) {
            imin[k+ks]=n+ns; dpmin[k+ks]=dp[ks][ns];
          }
       }
    }
   for (ks=0; ks < KS; ks++) {
      Q[imax[k+ks]]++;
      Q[imin[k+ks]]++;
    }
 }
```

The `forall` loop computes concurrently P dot-products, each dot-product requiring D multiplication/accumulations.

3 Architecture

3.1 Principle

The principle of the architecture is represented figure 1. It is composed of a matrix of $NS \times KS$ dot-product operators. Each dot-product is fed serially with the D the pixels and the skewers. As the pixels and the skewers are D-vector data, D cycles are required for computing P dot-products.

The results of the dot-product are stored into registers (shaded boxes) and shifted to KS MinMax units. These units compute the minimum and the maximum of the dot-product and kept trace of the pixel numbers which have produced these extrema values. The results of the MinMax units are shifted to the host processor through a fifo.

In order to get maximum performances, the minimum and maximum operations are performed in parallel with the dot-product computation: As soon as a dot-product phase is accomplished, the

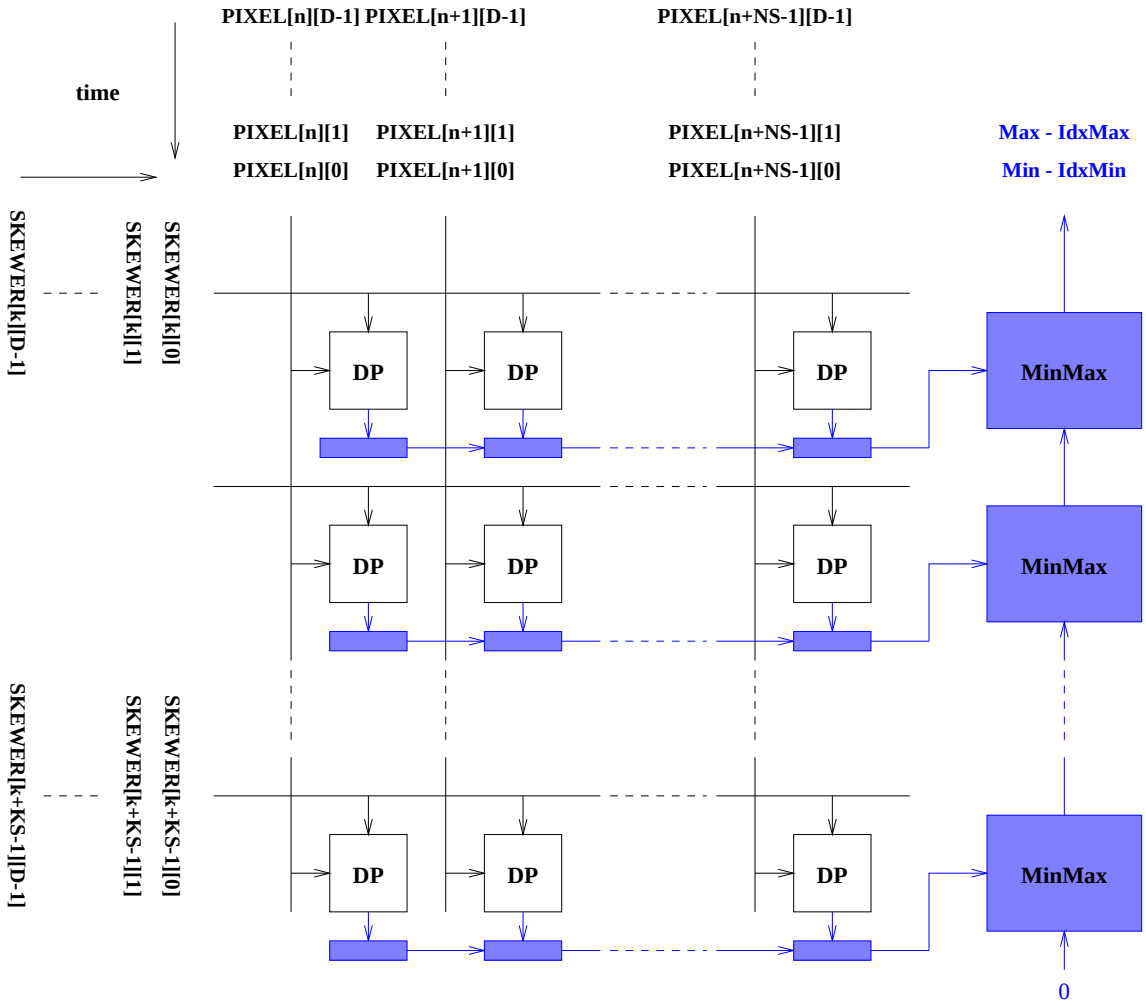


Figure 1: Architecture Principle

results are stored in the shift registers, and another phase begins immediately. During the next dot-product phase computation, the previous dot-product values are bit-serially shifted to the MinMax units. Hence, there is a complete overlap between the dot-product and the minimum/maximum computations.

Note that the updating of the Q array is not performed by the architecture. Since it represents a very small percentage of the overall computation, it is left to the host responsibility.

3.2 The Dot-Product Operator

The dot-product operator is an optimized 16-bit multiplier/accumulator. Pixels are 8-bit encoded (bits 13 to 5 from the initial 16-bit value) and skewers are 3-bit encoded. The skewer values range from -2 to +2. Consequently, the multiplication with a pixel by 0, 1 or 2 is straightforward: it is a null value, the pixel itself, or the pixel 1-bit left-shifted. The table below gives the skewer encoding:

Skewer Encoding			Corresponding
Skewer[2]	Skewer[1]	Skewer[0]	Decimal Value
0	0	0	0
0	0	1	-1
0	1	0	-2
0	1	1	x
1	0	0	x
1	0	1	1
1	1	0	2
1	1	1	x

Figure 2 details the architecture of the dot-product operator. The heart is an adder/subtractor-accumulator provided by the Xilinx logiblox library from the Alliance Series FPGA CAD tools [3]. Its input is connected to a multiplier unit which is actually reduced to a multiplexer gate for providing either a null value, a direct pixel or a 1-bit left-shifted pixel value. The `Reset` signal reset the accumulator to zero and load the shift register with the last computed dot-product.

The VHDL code of the Dot-Product operator is given in Annex A.

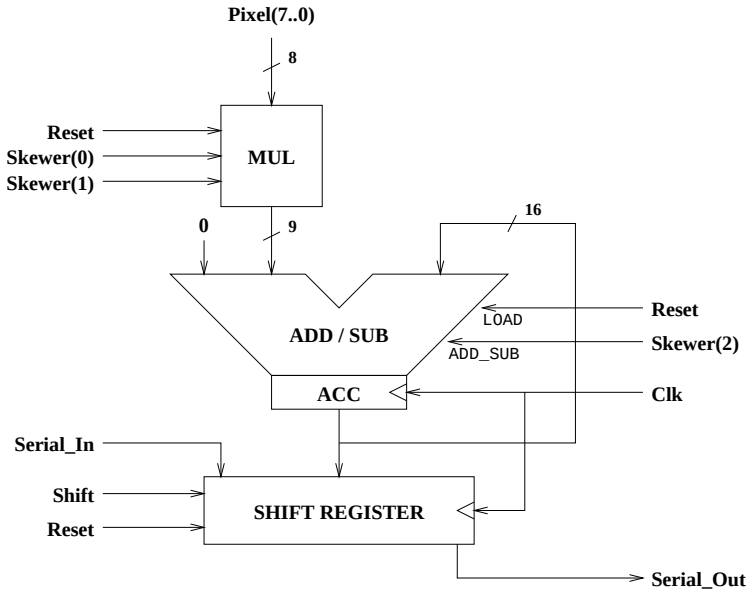


Figure 2: Dot-Product operator

3.3 The MinMax Unit

A MinMax unit receives bit-serially a stream of dot-product results. It has the charge of detecting the maximum and the minimum values together with the pixel index which has provided these extrema. It output serially two 32-bit data, first the maximum dot-product associated with its pixel index and, second the minimum dot-product and its associated pixel index. The lower 16 bits encode the min or max value and the upper 16 bits the pixel index. These 32-bit data are sent to the host.

The architecture is shown in figure 3. The `Buffer_in` register input serially 16 bits corresponding to a dot-product result. Thus, every 16 cycles, a comparison between the value stored in the `Buffer_in` register and the `AccMax` and `AccMin` registers is performed to update or not these registers.

After a complete cycle is achieved for computing the N dot-products against one particular skewer to every pixels, the maximum and minimum dot-products are sent to the host. The MinMax units are cascaded so that the host receives alternatively a maximum dot-product (with its pixel index) and a minimum dot-product (with its pixel index).

The input labeled `# Pixel` comes from a counter incremented each time a new dot-product is entered into the `Buffer_in` register. This counter is common to all the MinMax unit and computes the pixel index.

The VHDL code of a MinMax unit is given in annex B.

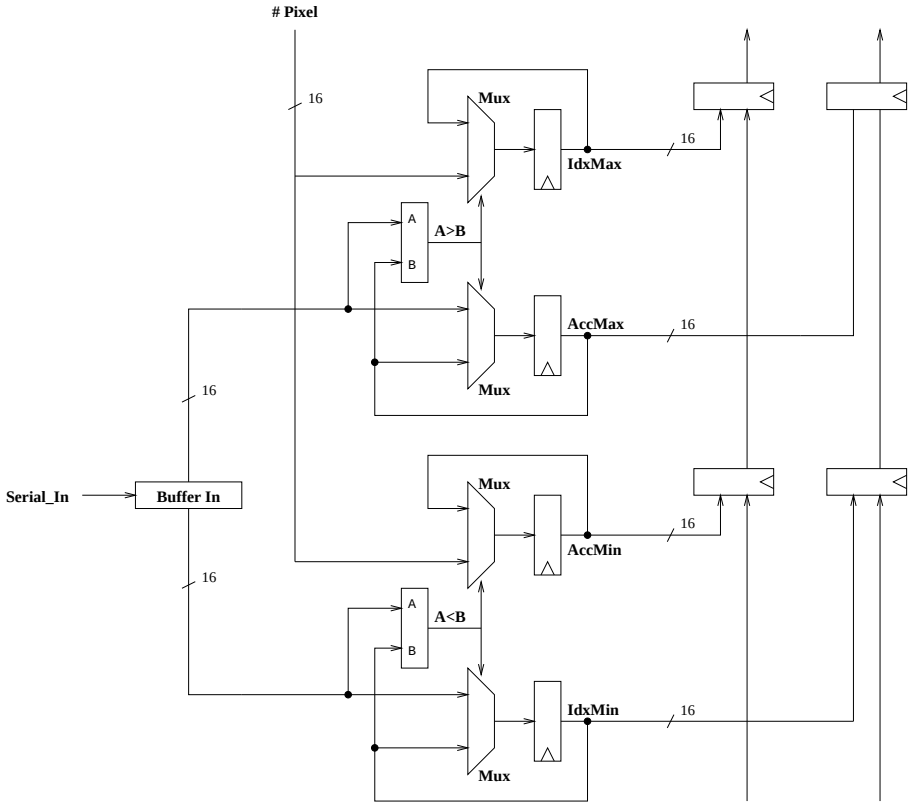


Figure 3: MinMax Unit

3.4 Control

There are two processes running concurrently. The first one controls the dot-product array (dot-product process), and the second one controls the MinMax units (minmax process). Both processes are implemented with a state machine as shown in Figure 4.

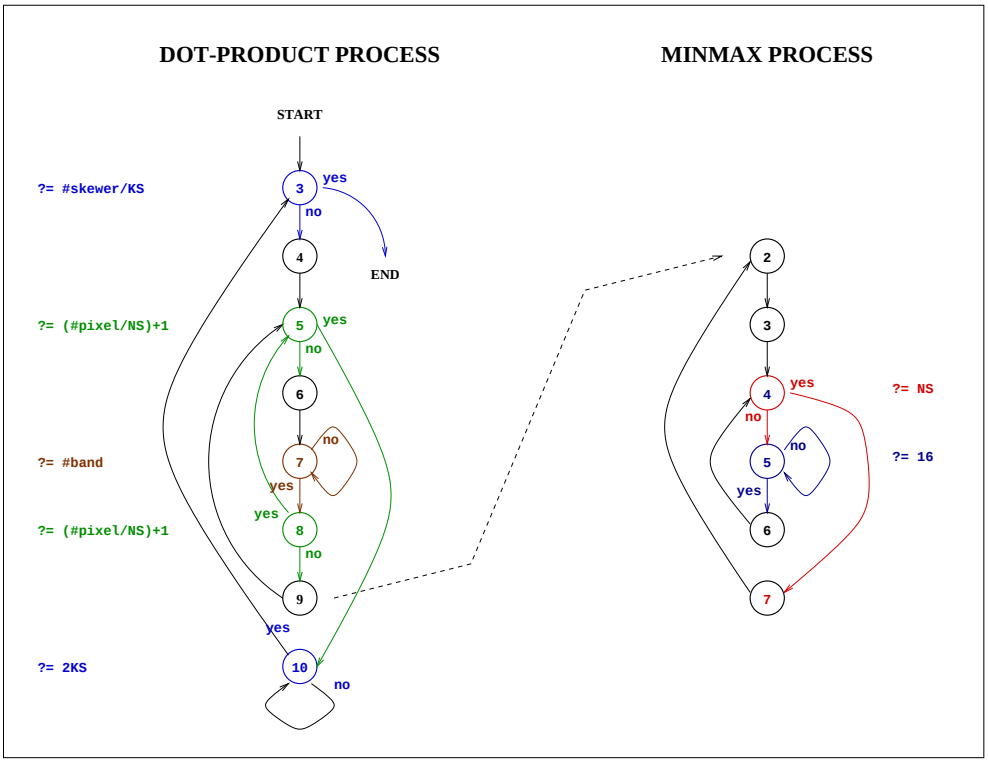


Figure 4: Control

The dot-product process is the main process. The outer loop performs (K/KS) iterations corresponding to the number of skewers (K) divided by the number of skewers concurrently computed (KS). Each iteration computes N dot-products (N = number of pixels). Knowing that NS pixels are processed in parallel, (N/NS) steps are required. Actually, and as it can be noticed on Figure 4, (N/NS)+1 steps are performed. This is due to the fact that the minmax computation of step i is achieved during the computation of step $(i+1)$. At the end of this step (except for the last one) the minmax process is activated (state 9 on figure 4).

The minmax process performs NS iterations. Each iteration first get serially KS different dot-products (16 cycles), then compute concurrently KS minimum and maximum according to these new dot-product values. To work properly, the time for computing a dot-product and the time for processing the results must respect the following constraint:

$$D >= (16 * NS)$$

The time for computing a dot-product (D cycles for a hyper-spectral image of D bands) must be greater (or at least equal) to the time for processing the min and max (precision of the results multiplied par the number of pixels processed concurrently).

At the end of each outer loop iteration, the dot-product process send KS results to the host. A result is composed of 2 pairs of (value,index) representing the maximum and minimum with their associated pixel index. Although further computation on the host to update the Q vector only

required the index, the maximum and minimum are also provided. Actually, they are needed for partitioning problem if the image or the skewers cannot fit in the board memory.

3.5 Partitioning

Partitioning is required if the reconfigurable board cannot store a complete hyper-spectral image (N D-vector pixels) and the K skewers in its memory. In that case, the computation has to be performed into several passes.

Let us assume that only K_p skewers and N_p pixels can be fit simultaneously to the board memory. The best way to minimize data transfer between the host and the board is to apply the PPI algorithm on sub-hyper-spectral images of N_p pixels, and for each sub-image to have K/K_p successive passes.

The partitioned algorithm is:

```
for (n=0; n<N/Np; n=++)
 {
   load_pixel (n*Np,(n+1)*Np);    // load a sub-image of Nb pixels
   for (k=0; k<K/Kp; k=k++)
    {
      load_skewer (k*Kp,(k+1)*Kp; // load Kp skewers
      PPI (Np,Kp);                // process PPI
    }
 }
```

In that partitioning scheme, an overhead, compared to the non-partitioned algorithm is introduced by reloading several times the skewers. But, actually, this overhead is very small: the PPI algorithm is a computed-bound problem and data transfer have a little of influence over the total execution time.

4 Implementation on the Wildforce Board

The Wildforce board is marketed by Annapolis Micro System Inc [1]. The board available at LANL is mainly composed of five Xilinx XC4036EX processing elements [2] interconnected in a 36-bit width systolic ring. Each processing element, except CPE0, is connected to a 512 Kbytes memory (128 K 32-bit words). The processing elements CPE0, PE1 and PE4 are connected to the PCI interface bus by a bidirectional 512 32-bit word fifos. The memories are dual port memories accessible both from the host (thru the PCI bus) and the processing element. A programmable crossbar connects all the processing elements.

The architecture described in the previous section (figure 5) is split into the four processing elements PE1, PE2, PE3 and PE4. CPE0 is not used. PE2, PE3 and PE4 house each 32 dot-product operators, leading to a total of 96 operators able to run concurrently. PE1 contains 8 MinMax

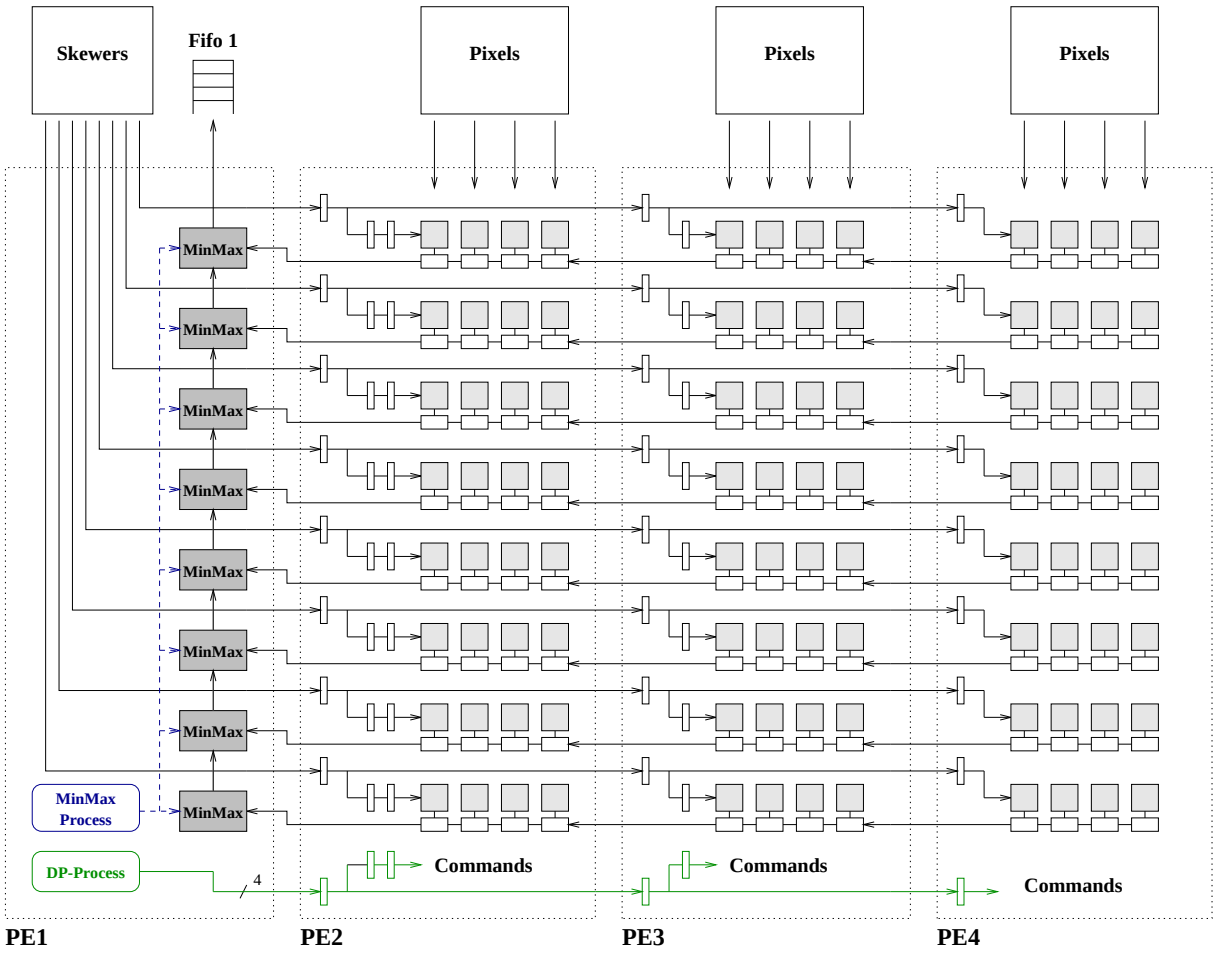
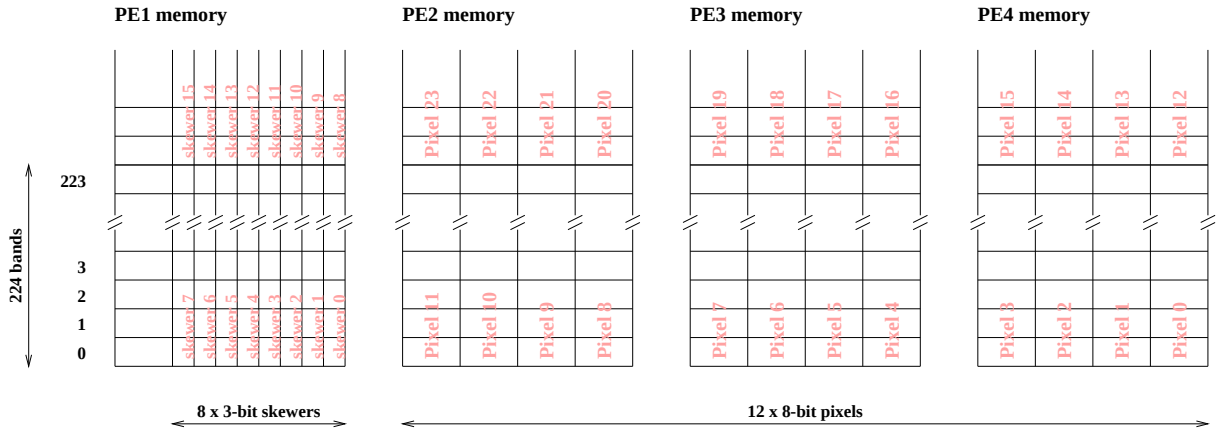


Figure 5: Wildforce Implementation

units and the two state machines. The memories of the processing elements PE2 to PE4 contain the pixels. The memory of PE1 contains the skewers.

The control located into the processing element PE1 is pipelined through the processing elements PE2 to PE4. The skewers, located into the memory of PE1, are propagated in the same way. The crossbar could have been used to broadcast the control and the skewers, avoiding this relatively complex control. There is one main reason for not using the crossbar: the internal PE pin connection does not fit with the layout organization of the architecture. The skewers have to flow "horizontally" while the crossbar in/out pins are provided "vertically". This results in long wires for connecting the skewers to the dot-product operators, leading to long delays and a low working frequency as experimented in some preliminary versions.

A sub hyper-spectral image is stored into the memory of the processing elements PE2 to PE4, while the memory of PE1 stores the skewers. The pixels and the skewers are stored as follows:



Since the size of a PE memory is 512 Kbytes, it can contain a maximum of 2340 224-band pixel. In other words, the board memory dedicated for storing a hyper-spectral image can hold a maximum of 7020 pixels. The current implementation stores 6144 pixels (512 × 12): each processor stores 2048 pixels and can simultaneously access 4 of them. Similarly, the PE1 memory stores 4096 skewers (512 × 8) and a single memory access provides 8 values.

The design has been described in VHDL, and synthesized with Synplify from Synplicity Inc. [4]. Manual placement has been performed to get a "regular" layout array in order to optimize the place-and-route step and obtain better performance. The design works at 25 MHz.

Performance

To determine the speed-up over a sequential programmable machine, the same algorithm has been coded in C and optimized accordingly. Tests have been made on real data provided by satellite remote sensors:

- image = 512 × 640 pixels
- 224 channels
- 4 000 skewers

A speed-up of 80 has been measured over a 450 MHz PC with 256 Mbytes SDRAM and running the LINUX system. In other words, a process taking 190 minutes can be reduced to 140 seconds (2 mn, 20 sec). The speed-up is calculated as the ration between the execution time (given by the unix command `time`) of the sequential algorithm and the wildforce implementation. Both include the time for accessing the data through the network.

References

[1] Wildforce Reference Manual, revision 3.4, Annapolis Micro System Inc, 1999 (www.annapmicro.com).

[2] Xilinx XC4000E and Xilinx XC4000X Series Field Programmable Gate Array, Product specification, version 1.6, Xilinx Inc, 1999 (www.xilinx.com).

[3] Alliance Series 2.1i, Xilinx Inc, 1999 (www.xilinx.com).

[4] Synplify User Guide, release 5.2.2, Synplicity Inc, 1999 (www.synplicity.com).

Annex A Dot-Product VHDL code

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mac2 is
 port
   (
     Clk       : in  std_logic;
     ClkEn     : in  std_logic;
     Reset     : in  std_logic;
     Shift     : in  std_logic;
     SerialIn  : in  std_logic;
     SerialOut : out std_logic;
     PixelIn   : in  std_logic_vector (7 downto 0);
     SkewerIn  : in  std_logic_vector (2 downto 0)
   );

end mac2;

architecture struct of mac2 is

component addsubacc
  PORT(
    ADD_SUB: IN std_logic;
    B      : IN std_logic_vector(15 DOWNTO 0);
    LOAD   : IN std_logic;
    CLK_EN : IN std_logic;
    CLOCK  : IN std_logic;
    Q_OUT  : OUT std_logic_vector(15 DOWNTO 0));
end component;

component mul_pix is
  port (
         SK0R : in std_logic;
         SK1  : in std_logic;
         Pi   : in std_logic;
         Px   : in std_logic;
         M    : out std_logic);
end component;

component Sk0Reset is
  port (
         Sk0  : in std_logic;
         Sk1  : in std_logic;
         Raz  : in std_logic;
         SK0R : out std_logic);
end component;
```

```
  signal Acc   : std_logic_vector (15 downto 0);
  signal Mul   : std_logic_vector (15 downto 0);
  signal Srg   : std_logic_vector (15 downto 0);
  signal SK0R  : std_logic;
  signal Zero  : std_logic;

begin

Zero <= '0';
SerialOut <= Srg(15);

asa0 : addsubacc port map
(ADD_SUB => SkewerIn(2),
 B       => Mul,
 LOAD    => Reset,
 CLK_EN  => ClkEn,
 CLOCK   => Clk,
 Q_OUT   => Acc);

skk: Sk0Reset port map (SkewerIn(0),SkewerIn(1),Reset,SK0R);
mp0: mul_pix port map (SK0R, SkewerIn(1),PixelIn(0),Zero,      Mul(0));
mp1: mul_pix port map (SK0R, SkewerIn(1),PixelIn(1),PixelIn(0),Mul(1));
mp2: mul_pix port map (SK0R, SkewerIn(1),PixelIn(2),PixelIn(1),Mul(2));
mp3: mul_pix port map (SK0R, SkewerIn(1),PixelIn(3),PixelIn(2),Mul(3));
mp4: mul_pix port map (SK0R, SkewerIn(1),PixelIn(4),PixelIn(3),Mul(4));
mp5: mul_pix port map (SK0R, SkewerIn(1),PixelIn(5),PixelIn(4),Mul(5));
mp6: mul_pix port map (SK0R, SkewerIn(1),PixelIn(6),PixelIn(5),Mul(6));
mp7: mul_pix port map (SK0R, SkewerIn(1),PixelIn(7),PixelIn(6),Mul(7));
mp8: mul_pix port map (SK0R, SkewerIn(1),Zero,      PixelIn(7),Mul(8));
Mul(15 downto 9) <= "0000000";

process (Clk)
  begin
    if rising_edge(Clk) then
      for i in 15 downto 1 loop
 Srg(i) <=    ( (Reset    ) and Acc(i) )
                  or ( (not Reset) and
                                      (    ( (Shift    ) and Srg(i-1) )
                                        or ( (not Shift) and Srg(i)   )
                                      )
                     );
 Srg(0) <=    ( (Reset    ) and Acc(0) )
                  or ( (not Reset) and
                                      (    ( (Shift    ) and SerialIn )
                                        or ( (not Shift) and Srg(0)   )
                                      )
                     );
      end loop;
    end if;
  end process;

end struct;
```

Annex B MinMax VHDL code

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity minmax is
 port
   (
     Clk      : in  std_logic;
     RazAcc   : in  std_logic;
     LoadAcc  : in  std_logic;
     LoadOut  : in  std_logic;
     ShiftIn  : in  std_logic;
     ShiftOut : in  std_logic;
     SerialIn : in  std_logic;
     IdxPix   : in  std_logic_vector (15  downto 0);
     DataIn   : in  std_logic_vector (31 downto 0);
     DataOut  : out std_logic_vector (31 downto 0)
   );
end minmax;

architecture struct of minmax is

component AinfB16
  PORT(
    A: IN std_logic_vector(15 DOWNTO 0);
    B: IN std_logic_vector(15 DOWNTO 0);
    A_LT_B: OUT std_logic);
end component;

component AsupB16
  PORT(
    A: IN std_logic_vector(15 DOWNTO 0);
    B: IN std_logic_vector(15 DOWNTO 0);
    A_GT_B: OUT std_logic);
end component;

  signal AccMin    : std_logic_vector (15 downto 0);
  signal AccMax    : std_logic_vector (15 downto 0);
  signal OutMin    : std_logic_vector (15 downto 0);
  signal OutMax    : std_logic_vector (15 downto 0);
  signal IdxAccMin : std_logic_vector (15 downto 0);
  signal IdxAccMax : std_logic_vector (15 downto 0);
  signal IdxOutMin : std_logic_vector (15 downto 0);
  signal IdxOutMax : std_logic_vector (15 downto 0);
  signal BuffIn    : std_logic_vector (15 downto 0);
  signal AsupB     : std_logic;
  signal AinfB     : std_logic;
```

```
begin

max : AsupB16 port map (BuffIn, AccMax, AsupB);
min : AinfB16 port map (BuffIn, AccMin, AinfB);

DataOut (31 downto 16) <= IdxOutMax;
DataOut (15 downto  0) <= OutMax;

process (Clk)
begin
  if rising_edge (Clk) then
            if RazAcc = '1' then
      AccMax <= (others => '0');
      AccMin <= (others => '1');
      Buffin <= (others => '0');
            else
      if ShiftIn = '1' then
        for i in 15 downto 1 loop
          BuffIn(i) <= BuffIn(i-1);
                end loop;
        BuffIn(0) <= SerialIn;
              end if;
      if LoadAcc = '1' then
if AsupB = '1' then
                  IdxAccMax <= IdxPix;
                  AccMax <= BuffIn;
                end if;
if AinfB = '1' then
                  IdxAccMin <= IdxPix;
                  AccMin <= BuffIn;
                end if;
      end if;
      if LoadOut = '1'  then
                OutMin <= AccMin;
        OutMax <= AccMax;
                IdxOutMin <= IdxAccMin;
        IdxOutMax <= IdxAccMax;
              end if;
      if ShiftOut = '1' then
        OutMax <= OutMin;
        IdxOutMax <= IdxOutMin;
                OutMin <= DataIn(15 downto 0);
                IdxOutMin <= DataIn(31 downto 16);
              end if;
    end if;
          end if;
        end process;

end struct;
```